—Séptima edición—

JAVA™ CÓMO PROGRAMAR



—Séptima edición—

JAVA™ CÓMO PROGRAMAR

—Séptima edición—

JAVA™ CÓMO PROGRAMAR

P. J. Deitel

Deitel & Associates, Inc.

H. M. Deitel

Deitel & Associates, Inc.

TRADUCCIÓN

Alfonso Vidal Romero Elizondo

Ingeniero en Sistemas Electrónicos

Instituto Tecnológico y de Estudios Superiores de Monterrey

Campus Monterrey

REVISIÓN TÉCNICA

Gabriela Azucena Campos García

Roberto Martínez Román

Departamento de Computación

Instituto Tecnológico y de Estudios Superiores de Monterrey

Campus Estado de México

Jorge Armando Aparicio Lemus

Coordinador del Área de Software

Universidad Tecnológica de El Salvador



México • Argentina • Brasil • Colombia • Costa Rica • Chile • Ecuador
España • Guatemala • Panamá • Perú • Puerto Rico • Uruguay • Venezuela

Datos de catalogación bibliográfica

Deitel, Paul J. y Harvey M. Deitel

Cómo programar en Java. Séptima edición

PEARSON EDUCACIÓN, México 2008
ISBN: 978-970-26-1190-5
Área: Computación

Formato: 20 × 25.5 cm Páginas: 1152

ISBN 0-13-222220-5

Edición en español

Editor: Luis Miguel Cruz Castillo
e-mail: luis.cruz@pearsoned.com
Editor de desarrollo: Bernardino Gutiérrez Hernández
Supervisor de producción: Enrique Trejo Hernández

Edición en inglés

Vice President and Editorial Director, ECS: ***Marcia J. Horton***
Associate Editor: ***Jennifer Cappello***
Assistant Editor: ***Carole Snyder***
Executive Managing Editor: ***Vince O'Brien***
Managing Editor: ***Bob Engelhardt***
Production Editors: ***Donna M. Crilly, Marta Samsel***
Director of Creative Services: ***Paul Belfanti***
A/V Production Editor: ***Xiaohong Zhu***
Art Studio: ***Artworks, York, PA***
Creative Director: ***Juan López***
Art Director: ***Kristine Carney***
Cover Design: ***Abbey S. Deitel, Harvey M. Deitel, Francesco Santalucia, Kristine Carney***
Interior Design: ***Harvey M. Deitel, Kristine Carney***
Manufacturing Manager: ***Alexis Heydt-Long***
Manufacturing Buyer: ***Lisa McDowell***
Executive Marketing Manager: ***Robin O'Brien***

SÉPTIMA EDICIÓN, 2008

Cámara Nacional de la Industria Editorial Mexicana. Reg. Núm. 1031.

ISBN 10: 970-26-1190-3
ISBN 13: 978-970-26-1190-5

Impreso en México. *Printed in Mexico*.
1 2 3 4 5 6 7 8 9 0 - 11 10 09 08



A Vince O'Brien,
Director de Administración de Proyectos, Prentice Hall.

Es un privilegio para nosotros trabajar con un profesional consumado.
Nuestros mejores deseos para tu éxito continuo.

Paul y Harvey

Marcas registradas

DEITEL, el insecto con dos pulgares hacia arriba y DIVE INTO son marcas registradas de Deitel and Associates, Inc.

Java y todas las marcas basadas en Java son marcas registradas de Sun Microsystems, Inc., en los Estados Unidos y otros países. Pearson Education es independiente de Sun Microsystems, Inc.

Microsoft, Internet Explorer y el logotipo de Windows son marcas registradas de Microsoft Corporation en los Estados Unidos y/o en otros países

UNIX es una marca registrada de The Open Group.

Contenido

Los capítulos 26 a 30 así como los apéndices, los encontrará en el CD que acompaña este libro.

Prefacio

"No vivas más en fragmentos, sólo conéctate".
—Edgar Morgan Foster

¡Bienvenido a Java y *Cómo programar en Java, 7ª edición*! En Deitel & Associates escribimos para Prentice Hall libros de texto sobre lenguajes de programación y libros de nivel profesional, impartimos capacitación a empresas en todo el mundo y desarrollamos negocios en Internet. Fue un placer escribir esta edición ya que refleja cambios importantes en el lenguaje Java y en las formas de impartir y aprender programación. Se han realizado ajustes considerables en todos los capítulos.

Características nuevas y mejoradas

He aquí una lista de las actualizaciones que hemos realizado a la 6ª y 7ª ediciones:

- Actualizamos todo el libro a la nueva plataforma Java Standard Edition 6 ("Mustang") y lo revisamos cuidadosamente, en base a la *Especificación del lenguaje Java*.
- Revisamos la presentación conforme a las recomendaciones del currículum de ACM/IEEE.
- Reforzamos nuestra pedagogía anticipada sobre las clases y los objetos, poniendo especial atención a la orientación de los profesores universitarios en nuestros equipos de revisión, para asegurarnos de obtener el nivel conceptual correcto. Todo el libro está orientado a objetos, y las explicaciones sobre la POO son claras y accesibles. En el capítulo 1 presentamos los conceptos básicos y la terminología de la tecnología de objetos. Los estudiantes desarrollan sus primeras clases y objetos personalizados en el capítulo 3. Al presentar los objetos y las clases en los primeros capítulos, hacemos que los estudiantes "piensen acerca de objetos" de inmediato, y que dominen estos conceptos con más profundidad.
- La primera presentación de clases y objetos incluye los ejemplos prácticos de las clases `Tiempo`, `Empleado` y `LibroCalificaciones`, los cuales van haciendo su propio camino a través de varias secciones y capítulos, presentando conceptos de OO cada vez más profundos.
- Los profesores que imparten cursos introductorios tienen una amplia opción en cuanto a la cantidad de GUI y gráficos a cubrir; desde cero, a una secuencia introductoria de diez secciones breves, hasta un tratamiento detallado en los capítulos 11, 12 y 22, y en el apéndice F.
- Adaptamos nuestra presentación orientada a objetos para utilizar la versión más reciente de *UML*™ (*Lenguaje Unificado de Modelado*™): *UML*™ *2*, el lenguaje gráfico estándar en la industria para modelar sistemas orientados a objetos.
- En los capítulos 1-8 y 10 presentamos y adaptamos el ejemplo práctico opcional del cajero automático (ATM) de DOO/UML 2. Incluimos un apéndice Web adicional, con la implementación completa del código. Dé un vistazo a los testimonios que se incluyen en la parte posterior del libro.
- Agregamos varios ejemplos prácticos sustanciales sobre programación Web orientada a objetos.
- Actualizamos el capítulo 25, Acceso a bases de datos con JDBC, para incluir JDBC 4 y utilizar el nuevo sistema de administración de bases de datos Java DB/Apache Derby, además de MySQL. Este capítulo incluye un ejemplo práctico OO sobre el desarrollo de una libreta de direcciones controlada por una base de datos, la cual demuestra las instrucciones preparadas y el descubrimiento automático de controladores de JDBC 4.
- Agregamos los capítulos 26 y 27, Aplicaciones Web: partes 1 y 2, que introducen la tecnología JavaServer Faces (JSF) y la utilizan con Sun Java Studio Creador 2 para construir aplicaciones Web de una manera rápida y sencilla. El capítulo 26 incluye ejemplos sobre la creación de GUIs de aplicaciones Web,

el manejo de eventos, la validación de formularios y el rastreo de sesiones. El material de JSF sustituye los capítulos anteriores sobre servlets y JavaServer Pages (JSP).

- Agregamos el capítulo 27, Aplicaciones Web: parte 2, que habla acerca del desarrollo de aplicaciones Web habilitadas para Ajax, usando las tecnologías JavaServer Faces y Java BluePrints. Este capítulo incluye una aplicación de libreta de direcciones Web multiniveles, controlada por una base de datos, que permite a los usuarios agregar y buscar contactos, y mostrar las direcciones de los contactos en mapas de Google™ Maps. Esta aplicación habilitada para Ajax le proporciona una sensación real del desarrollo Web 2.0. La aplicación utiliza Componentes JSF habilitados para Ajax para sugerir los nombres de los contactos, mientras el usuario escribe un nombre para localizar y mostrar una dirección localizada en un mapa de Google Maps.
- Agregamos el capítulo 28, Servicios Web JAX-WS, Web 2.0 y Mash-ups que utiliza un método basado en herramientas para crear y consumir servicios Web, una capacidad típica de Web 2.0. Los ejemplos prácticos incluyen el desarrollo de los servicios Web del juego de blackjack y un sistema de reservaciones de una aerolínea.
- Utilizamos el nuevo método basado en herramientas para desarrollar aplicaciones Web con rapidez; todas las herramientas pueden descargarse sin costo.
- Fundamos la Iniciativa Deitel de Negocios por Internet (Deitel Internet Business Initiative) con 60 nuevos centros de recursos para apoyar a nuestros lectores académicos y profesionales. Dé un vistazo a nuestros nuevos centros de recursos (`www.deitel.com/resourcecenters.html`), incluyendo: Java SE 6 (Mustang), Java, Evaluación y Certificación de Java, Patrones de Diseño de Java, Java EE 5, Motores de Búsqueda de Código y Sitios de Código, Programación de Juegos, Proyectos de Programación y muchos más. Regístrese en el boletín de correo electrónico gratuito *Deitel® Buzz Online* (`www.deitel.com/newsletter/subscribe.html`); cada semana anunciamos nuestro(s) centro(s) de recurso(s) más reciente(s); además incluimos otros temas de interés para nuestros lectores.
- Hablamos sobre los conceptos clave de la comunidad de ingeniería de software, como Web 2.0, Ajax, SOA, servicios Web, software de código fuente abierto, patrones de diseño, mashups, refabricación, programación extrema, desarrollo ágil de software, prototipos rápidos y mucho más.
- Rediseñamos por completo el capítulo 23, Subprocesamiento múltiple [nuestro agradecimiento especial a Brian Goetz y Joseph Bowbeer, coautores de *Java Concurrency in Practice*, Addison-Wesley, 2006].
- Hablamos sobre la nueva clase `SwingWorker` para desarrollar interfaces de usuario con subprocesamiento múltiple.
- Hablamos sobre los nuevos Componentes de Integración de Escritorio de Java (JDIC), como las pantallas de inicio (splash screens) y las interacciones con la bandeja del sistema.
- Hablamos sobre el nuevo administrador de esquemas `GroupLayout` en el contexto de la herramienta de diseño de GUI NetBeans 5.5 Matisse para crear GUIs portables que se adhieran a los lineamientos de diseño de GUI de la plataforma subyacente.
- Presentamos las nuevas características de ordenamiento y filtrado de `JTable`, que permiten al usuario reordenar los datos en un objeto `JTable` y filtrarlos mediante expresiones regulares.
- Presentamos un tratamiento detallado de los genéricos y las colecciones de genéricos.
- Introducimos los mashups, aplicaciones que, por lo general, se crean mediante llamadas a servicios Web (y/o usando fuentes RSS) de dos o más sitios; otra característica típica de Web 2.0.
- Hablamos sobre la nueva clase `StringBuilder`, que tiene un mejor desempeño que `StringBuffer` en aplicaciones sin subprocesamiento.
- Presentamos las anotaciones, que reducen en gran parte la cantidad de código necesario para crear aplicaciones.

Las características que se presentan en Cómo programar en Java, 7a edición, incluyen:

- Cómo obtener entrada con formato mediante la clase `Scanner`.
- Mostrar salida con formato mediante el método `printf` del objeto `System.out`.

- Instrucciones `for` mejoradas para procesar elementos de arreglos y colecciones.
- Declaración de métodos con listas de argumentos de longitud variable ("varargs").
- Uso de clases `enum` que declaran conjuntos de constantes.
- Importación de los miembros `static` de una clase para usarlos en otra.
- Conversión de valores de tipo primitivo a objetos de envolturas de tipo y viceversa, usando autoboxing y auto-unboxing, respectivamente.
- Uso de genéricos para crear modelos generales de métodos y clases que pueden declararse una vez, pero usarse con muchos tipos de datos distintos.
- Uso de las estructuras de datos mejoradas para genéricos de la API Collections.
- Uso de la API Concurrency para implementar aplicaciones con subprocesamiento múltiple.
- Uso de objetos `RowSet` de JDBC para acceder a los datos en una base de datos.

Todo esto ha sido revisado cuidadosamente por distinguidos profesores y desarrolladores de la industria, que trabajaron con nosotros en *Cómo programar en Java 6ª* y *7ª ediciones*.

Creemos que este libro y sus materiales de apoyo proporcionarán a los estudiantes y profesionales una experiencia informativa, interesante, retadora y placentera. El libro incluye una extensa suite de materiales complementarios para ayudar a los profesores a maximizar la experiencia de aprendizaje de sus estudiantes.

Cómo programar en Java 7ª edición presenta cientos de programas completos y funcionales, y describe sus entradas y salidas. Éste es nuestro característico método de "código activo" ("live code"); presentamos la mayoría de los conceptos de programación de Java en el contexto de programas funcionales completos.

Si surge alguna duda o pregunta a medida que lee este libro, envíe un correo electrónico a `deitel@deitel.com`; le responderemos a la brevedad. Para obtener actualizaciones sobre este libro y el estado de todo el software de soporte de Java, además de las noticias más recientes acerca de todas las publicaciones y servicios de Deitel, visite `www.deitel.com`.

Regístrese en `www.deitel.com/newsletter/subscribe.html` para obtener el boletín de correo electrónico *Deitel® Buzz Online* y visite la página `www.deitel.com/resourcecenters.html` para tener acceso a nuestra lista creciente de centros de recursos.

Uso de UML 2 para desarrollar un diseño orientado a objetos de un ATM. UML 2 se ha convertido en el lenguaje de modelado gráfico preferido para diseñar sistemas orientados a objetos. Todos los diagramas de UML en el libro cumplen con la especificación UML 2. Utilizamos los diagramas de actividad de UML para demostrar el flujo de control en cada una de las instrucciones de control de Java, y usamos los diagramas de clases de UML para representar las clases y sus relaciones de herencia en forma visual.

Incluimos un ejemplo práctico opcional (pero altamente recomendado) acerca del diseño orientado a objetos mediante el uso de UML. La revisión del ejemplo práctico estuvo a cargo de un distinguido equipo de profesores y profesionales de la industria relacionados con DOO/UML, incluyendo líderes en el campo de Rational (los creadores de UML) y el Grupo de administración de objetos (responsable de la evolución de UML). En el ejemplo práctico, diseñamos e implementamos por completo el software para un cajero automático (ATM) simple. Las secciones Ejemplo práctico de Ingeniería de Software al final de los capítulos 1 a 8 y 10 presentan una introducción cuidadosamente planeada al diseño orientado a objetos mediante el uso de UML. Presentamos un subconjunto conciso y simplificado de UML 2, y después lo guiamos a través de su primera experiencia de diseño, ideada para los principiantes. El ejemplo práctico no es un ejercicio, sino una experiencia de aprendizaje de principio a fin, que concluye con un recorrido detallado a través del código completo en Java. Las secciones del Ejemplo Práctico de Ingeniería de Software ayudan a los estudiantes a desarrollar un diseño orientado a objetos para complementar los conceptos de programación orientada a objetos que empiezan a aprender en el capítulo 1, y que implementan en el capítulo 3. En la primera de estas secciones, al final del capítulo 1, introducimos los conceptos básicos y la terminología del DOO. En las secciones opcionales Ejemplo Práctico de Ingeniería de Software al final de los capítulos 2 a 5, consideramos cuestiones más sustanciales al emprender la tarea de resolver un problema retador con las técnicas del DOO. Analizamos un documento de requerimientos típico que especifica un sistema a construir, determina los objetos necesarios para implementar ese sistema, establece los atributos que deben tener estos objetos, fija los comportamientos que deben exhibir estos objetos y especifica la forma en que deben interactuar los objetos entre sí para cumplir con los requerimientos del sistema. En un apéndice

Web adicional presentamos el código completo de una implementación en Java del sistema orientado a objetos que diseñamos en los primeros capítulos. El ejemplo práctico ayuda a preparar a los estudiantes para los tipos de proyectos sustanciales que encontrarán en la industria. Empleamos un proceso de diseño orientado a objetos cuidadosamente desarrollado e incremental para producir un modelo en UML 2 para nuestro sistema ATM. A partir de este diseño, producimos una implementación sustancial funcional en Java, usando las nociones clave de la programación orientada a objetos, incluyendo clases, objetos, encapsulamiento, visibilidad, composición, herencia y polimorfismo.

Gráfico de dependencias

En el gráfico de la siguiente página se muestra las dependencias entre los capítulos, para ayudar a los profesores a planear su programa de estudios. *Cómo programar en Java 7ª edición* es un libro extenso, apropiado para una variedad de cursos de programación en distintos niveles. Los capítulos 1-14 forman una secuencia de programación elemental accesible, con una sólida introducción a la programación orientada a objetos. Los capítulos 11, 12, 20, 21 y 22 forman una secuencia sustancial de GUI, gráficos y multimedia. Los capítulos 15 a 19 forman una excelente secuencia de estructuras de datos. Los capítulos 24 a 28 forman una clara secuencia de desarrollo Web con uso intensivo de bases de datos.

Método de enseñanza

Cómo programar en Java 7ª edición contiene una extensa colección de ejemplos. El libro se concentra en los principios de la buena ingeniería de software, haciendo hincapié en la claridad de los programas. Enseñamos mediante ejemplos. Somos educadores que impartimos temas de vanguardia en salones de clases de la industria alrededor del mundo. El Dr. Harvey M. Deitel tiene 20 años de experiencia en la enseñanza universitaria y 17, en la enseñanza en la industria. Paul Deitel tiene 15 años de experiencia en la enseñanza en la industria. Juntos han impartido cursos, en todos los niveles, a clientes gubernamentales, industriales, militares y académicos de Deitel & Associates.

Método del código activo. *Cómo programar en Java 7ª edición* está lleno de ejemplos de "código activo"; esto significa que cada nuevo concepto se presenta en el contexto de una aplicación en Java completa y funcional, que es seguido inmediatamente por una o más ejecuciones actuales, que muestran las entradas y salidas del programa. Este estilo ejemplifica la manera en que enseñamos y escribimos acerca de la programación; a éste le llamamos el método del "código activo".

Resaltado de código. Colocamos rectángulos de color gris alrededor de los segmentos de código clave en cada programa.

Uso de fuentes para dar énfasis. Colocamos los términos clave y la referencia a la página del índice para cada ocurrencia de definición en texto **en negritas** para facilitar su referencia. Enfatizamos los componentes en pantalla en la fuente **Helvética en negritas** (por ejemplo, el menú **Archivo**) y enfatizamos el texto del programa en la fuente `Lucida` (por ejemplo, `int x = 5`).

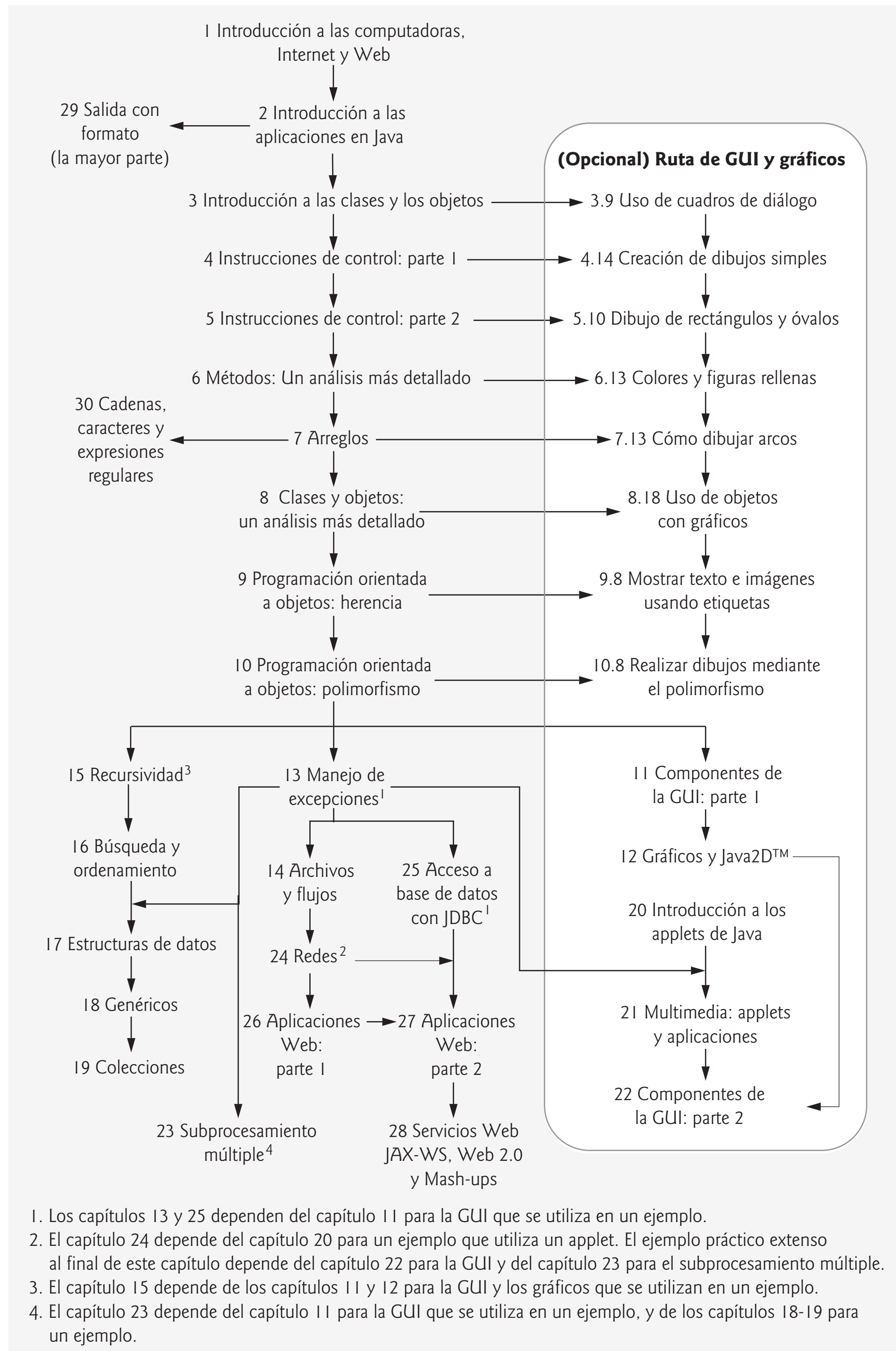
Acceso Web. Todos los ejemplos de código fuente para *Cómo programar en Java 7ª edición* (y para nuestras otras publicaciones) se pueden descargar en:

```
www.deitel.com/books/jhtp7
www.pearsoneducacion.net/deitel
```

El registro en el sitio es un proceso fácil y rápido. Descargue todos los ejemplos y, a medida que lea las correspondientes discusiones en el libro de texto, después ejecute cada programa. Realizar modificaciones a los ejemplos y ver los efectos de esos cambios es una excelente manera de mejorar su experiencia de aprendizaje en Java.

Objetivos. Cada capítulo comienza con una declaración de objetivos. Esto le permite saber qué es lo que debe esperar y le brinda la oportunidad, después de leer el capítulo, de determinar si ha cumplido con ellos.

Frases. Después de los objetivos de aprendizaje aparecen una o más frases. Algunas son graciosas, otras filosóficas y las demás ofrecen ideas interesantes. Esperamos que disfrute relacionando las frases con el material del capítulo.



Plan general. El plan general de cada capítulo le permite abordar el material de manera ordenada, para poder anticiparse a lo que está por venir y establecer un ritmo cómodo y efectivo de aprendizaje.

Ilustraciones/Figuras. Incluimos una gran cantidad de gráficas, tablas, dibujos lineales, programas y salidas de programa. Modelamos el flujo de control en las instrucciones de control mediante diagramas de actividad en

UML. Los diagramas de clases de UML modelan los campos, constructores y métodos de las clases. En el ejemplo práctico opcional del ATM de DOO/UML 2 hacemos uso extensivo de seis tipos principales de diagramas en UML.

Tips de programación. Incluimos tips de programación para ayudarle a enfocarse en los aspectos importantes del desarrollo de programas. Estos tips y prácticas representan lo mejor que hemos podido recabar a lo largo de seis décadas combinadas de experiencia en la programación y la enseñanza. Una de nuestras alumnas, estudiante de matemáticas, recientemente nos comentó que siente que este método es similar al de resaltar axiomas, teoremas y corolarios en los libros de matemáticas, ya que proporciona una base sólida sobre la cual se puede construir buen software.



Buena práctica de programación

Las buenas prácticas de programación *llaman la atención hacia técnicas que le ayudarán a producir programas más claros, comprensibles y fáciles de mantener.*



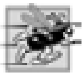
Error común de programación

Con frecuencia, los estudiantes tienden a cometer ciertos tipos de errores; al poner atención en estos Errores comunes de programación *se reduce la probabilidad de que usted pueda cometerlos.*



Tip para prevenir errores

Estos tips contienen sugerencias para exponer los errores y eliminarlos de sus programas; muchos de ellos describen aspectos de Java que evitan que los errores entren a los programas.



Tip de rendimiento

A los estudiantes les gusta "turbo cargar" sus programas. Estos tips resaltan las oportunidades para hacer que sus programas se ejecuten más rápido, o para minimizar la cantidad de memoria que ocupan.



Tip de portabilidad

Incluimos Tips de portabilidad *para ayudarle a escribir el código que pueda ejecutarse en una variedad de plataformas, y que expliquen cómo es que Java logra su alto grado de portabilidad.*



Observación de ingeniería de software

Las Observaciones de ingeniería de software *resaltan los asuntos de arquitectura y diseño, lo cual afecta la construcción de los sistemas de software, especialmente los de gran escala.*



Observaciones de apariencia visual

Le ofrecemos Observaciones de apariencia visual *para resaltar las convenciones de la interfaz gráfica de usuario. Estas observaciones le ayudan a diseñar interfaces gráficas de usuario atractivas y amigables para el usuario, en conformidad con las normas de la industria.*

Sección de conclusión. Cada uno de los capítulos termina con una sección breve de "conclusión", que recapitula el contenido del capítulo y la transición al siguiente capítulo.

Viñetas de resumen. Cada capítulo termina con estrategias pedagógicas adicionales. Presentamos un resumen detallado del capítulo, estilo lista con viñetas, sección por sección.

Terminología. Incluimos una lista alfabetizada de los términos importantes definidos en cada capítulo.

Ejercicios de autoevaluación y respuestas. Se incluyen diversos ejercicios de autoevaluación con sus respuestas, para que los estudiantes practiquen por su cuenta.

Ejercicios. Cada capítulo concluye con un diverso conjunto de ejercicios, incluyendo recordatorios simples de terminología y conceptos importantes; identificar los errores en muestras de código, escribir instrucciones individuales de programas; escribir pequeñas porciones de métodos y clases en Java; escribir métodos, clases y programas completos; y crear proyectos finales importantes. El extenso número de ejercicios permite a los instructores adaptar sus cursos a las necesidades únicas de sus estudiantes, y variar las asignaciones de los cursos cada semestre. Los profesores pueden usar estos ejercicios para formar tareas, exámenes cortos, exámenes regulares y proyectos finales. [NOTA: **No nos escriba para solicitarnos acceso al Centro de Recursos para Instructores. El acceso está limitado estrictamente a profesores universitarios que impartan clases en base al libro. Los profesores sólo pueden obtener acceso a través de los representantes de Pearson Educación].** Asegúrese de revisar nuestro centro de recursos de proyectos de programación (`http://www.deitel.com/ProgrammingProjects/`) para obtener muchos ejercicios adicionales y posibilidades de proyectos.

Miles de entradas en el índice. Hemos incluido un extenso índice, que es útil, en especial, cuando se utiliza el libro como referencia.

"Doble indexado" de ejemplos de código activo de Java. Para cada programa de código fuente en el libro, indexamos la leyenda de la figura en forma alfabética y como subíndice, bajo "Ejemplos". Esto facilita encontrar los ejemplos usando las características especiales.

Recursos para el estudiante incluidos en *Cómo programar en Java 7ª edición*

Hay, disponibles a la venta, una variedad de herramientas de desarrollo, pero ninguna de ellas es necesaria para comenzar a trabajar con Java. Escribimos *Cómo programar en Java 7ª edición* utilizando sólo el nuevo Kit de Desarrollo de Java Standard Edition (JDK), versión 6.0. Puede descargar la versión actual del JDK del sitio Web de Java de Sun: `java.sun.com/javase/downloads/index.jsp`. Este sitio también contiene las descargas de la documentación del JDK.

El CD que se incluye con este libro contienen el Entorno de Desarrollo Integrado (IDE) NetBeans™ 5.5 para desarrollar todo tipo de aplicaciones en Java, y el software Sun Java™ Studio Creator 2 Update 1 para el desarrollo de aplicaciones Web. Se proporciona también una versión en Windows de MySQL® 5.0 Community Edition 5.0.27 y MySQL Connector/J 5.0.4, para el procesamiento de datos que se lleva a cabo en los capítulos 25 a 28.

El CD también contiene los ejemplos del libro y una página Web con vínculos al sitio Web de Deitel & Associates, Inc. Puede cargar esta página Web en un explorador Web para obtener un rápido acceso a todos los recursos.

Encontrará recursos adicionales y descargas de software en nuestro centro de recursos de Java SE 6 (Mustang), ubicado en:

`www.deitel.com/JavaSE6Mustang/`

Java Multimedia Cyber Classroom 7ª edición

Cómo programar en Java 7ª edición incluye multimedia interactiva con mucho audio y basada en Web, complementaria para el libro *Java Multimedia Cyber Classroom, 7ª edición*, disponible en inglés. Nuestro *Ciber salón de clases* (*Cyber Classroom*) basado en Web incluye recorridos con audio de los ejemplos de código de los capítulos 1 a 14, soluciones a casi la mitad de los ejercicios del libro, un manual de laboratorio y mucho más. Para obtener más información acerca del *Cyber Classroom* basado en Web, visite:

`www.prenhall.com/deitel/cyberclassroom/`

A los estudiantes que utilizan nuestros *Ciber salones de clases* les gusta su interactividad y capacidades de referencia. Los profesores nos dicen que sus estudiantes disfrutan al utilizar el *Ciber salón de clases* y, en consecuencia, invierten más tiempo en los cursos, dominando un porcentaje mayor del material que en los cursos que sólo utilizan libros de texto.

Recursos para el instructor de *Cómo programar en Java 7ª edición*

Cómo programar en Java 7ª edición tiene una gran cantidad de recursos para los profesores. El *Centro de Recursos para Instructores* de Prentice Hall contiene el *Manual de soluciones*, con respuestas para la mayoría de los ejercicios al final de cada capítulo, un *Archivo de elementos de prueba* de preguntas de opción múltiple (aproximadamente dos por cada sección del libro) y diapositivas en PowerPoint® que contienen todo el código y las figuras del texto, además de los elementos en viñetas que sintetizan los puntos clave del libro. Los profesores pueden personalizar las diapositivas. Si usted todavía no es un miembro académico registrado, póngase en contacto con su representante de Pearson Educación. Cabe mencionar que todos estos recursos se encuentran en inglés.

Boletín de correo electrónico gratuito *Deitel® Buzz Online*

Cada semana, el boletín de correo electrónico *Deitel® Buzz Online* anuncia nuestro(s) centro(s) de recursos más reciente(s) e incluye comentarios acerca de las tendencias y desarrollos en la industria, vínculos a artículos y recursos gratuitos de nuestros libros publicados y de las próximas publicaciones, itinerarios de lanzamiento de productos, fe de erratas, retos, anécdotas, información sobre nuestros cursos de capacitación corporativa impartidos por instructores y mucho más. También es una buena forma para que usted se mantenga actualizado acerca de todo lo relacionado con *Cómo programar en Java 7ª edición*. Para suscribirse, visite la página Web:

`www.deitel.com/newsletter/subscribe.html`

Novedades en Deitel

Centros de recursos y la iniciativa de negocios por Internet de Deitel. Hemos creado muchos centros de recursos en línea (en `www.deitel.com/resourcecenters.html`) para mejorar su experiencia de aprendizaje en Java. Anunciamos nuevos centros de recursos en cada edición del boletín de correo electrónico *Deitel® Buzz Online*. Aquellos de especial interés para los lectores de este libro incluyen: Java, Certificación en Java, Patrones de Diseño en Java, Java EE 5, Java SE 6, AJAX, Apache, Motores de Búsqueda de Código y Sitios de Código, Eclipse, Programación de Juegos, Mashups, MySQL, Código Abierto, Proyectos de Programación, Web2.0, Web 3.0, Servicios Web y XML. Los centros de recursos de Deitel adicionales incluyen: Programas Afiliados, Servicios de Alerta, ASP.NET, Economía de Atención, Creación de Comunidades Web, C, C++, C#, Juegos de Computadora, DotNetNuke, FireFox, Gadgets, Google AdSense, Google Analytics, Google Base, Google Services, Google Video, Google Web Toolkit, IE7, Iniciativa de Negocios por Internet, Publicidad por Internet, Internet Video, Linux, Microformatos, .NET, Ning, OpenGL, Perl, PHP, Podcasting, Python, Recommender Systems, RSS, Ruby, Motores de Búsqueda, Optimización de Motores de Búsqueda, Skype, Sudoku, Mundos Virtuales, Visual Basic, Wikis, Windows Vista, WinFX y muchos más por venir.

Iniciativa de contenido libre. Nos complace ofrecerle artículos de invitados y tutoriales gratuitos, seleccionados de nuestras publicaciones actuales y futuras como parte de nuestra iniciativa de contenido libre. En cada tema del boletín de correo electrónico *Deitel® Buzz Online*, anunciamos las adiciones más recientes a nuestra biblioteca de contenido libre.

Reconocimientos

Uno de los mayores placeres al escribir un libro de texto es el de reconocer el esfuerzo de mucha gente, cuyos nombres quizá no aparezcan en la portada, pero cuyo arduo trabajo, cooperación, amistad y comprensión fue crucial para la elaboración de este libro. Mucha gente en Deitel & Associates, Inc. dedicó largas horas a este proyecto; queremos agradecer en especial a Abbey Deitel y Barbara Deitel.

También nos gustaría agradecer a dos participantes de nuestro programa de Pasantía con Honores, que contribuyeron a esta publicación: Megan Shuster, con especialidad en ciencias computacionales en el Swarthmore College, y Henry Klementowicz, con especialidad en ciencias computacionales en la Universidad de Columbia.

Nos gustaría mencionar nuevamente a nuestros colegas que realizaron contribuciones importantes a *Cómo programar en Java 6ª edición*: Andrew B. Goldberg, Jeff Listfield, Su Zhang, Cheryl Yaeger, Jing Hu, Sin Han Lo, John Paul Casiello y Christi Kelsey.

Somos afortunados al haber trabajado en este proyecto con un talentoso y dedicado equipo de editores profesionales en Prentice Hall. Apreciamos el extraordinario esfuerzo de Marcia Horton, Directora Editorial de la División de Ingeniería y Ciencias Computacionales de Prentice Hall. Jennifer Cappello y Dolores Mars hicieron

un excelente trabajo al reclutar el equipo de revisión del libro y administrar el proceso de revisión. Francesco Santalucia (un artista independiente) y Kristine Carney de Prentice Hall hicieron un maravilloso trabajo al diseñar la portada del libro; nosotros proporcionamos el concepto y ellos lo hicieron realidad. Vince O'Brien, Bob Engelhardt, Donna Crilly y Marta Samsel hicieron un extraordinario trabajo al administrar la producción del libro.

Deseamos reconocer el esfuerzo de nuestros revisores. Al adherirse a un estrecho itinerario, escrutinizaron el texto y los programas, proporcionando innumerables sugerencias para mejorar la precisión e integridad de la presentación.

Apreciamos con sinceridad los esfuerzos de nuestros revisores de post-publicación de la 6ª edición, y nuestros revisores de la 7ª edición:

Revisores de Cómo programar en Java 7ª edición (incluyendo los revisores de la post-publicación de la 6ª edición)

Revisores de Sun Microsystems: Lance Andersen (Líder de especificaciones de JDBC/Rowset, Java SE Engineering), Ed Burns, Ludovic Champenois (Servidor de Aplicaciones de Sun para programadores de Java EE con Sun Application Server y herramientas: NetBeans, Studio Enterprise y Studio Creador), James Davidson, Vadiraj Deshpande (Grupo de Integración de Sistemas de Java Enterprise, Sun Microsystems India), Sanjay Dhamankar (Grupo Core Developer Platform), Jesse Glick (Grupo NetBeans), Brian Goetz (autor de *Java Concurrency in Practice*, Addison-Wesley, 2006), Doug Kohlert (Grupo Web Technologies and Standards), Sandeep Konchady (Organización de Ingeniería de Software de Java), John Morrison (Grupo Portal Server Product de Sun Java System), Winston Prakash, Brandon Taylor (grupo SysNet dentro de la División de Software) y Jayashri Visvanathan (Equipo de Java Studio Creador de Sun Microsystems). **Revisores académicos y de la industria:** Akram Al-Rawi (Universidad King Faisal), Mark Biamonte (DataDiret), Ayad Boudiab (Escuela Internacional de Choueifat, Líbano), Joe Bowbeer (Mobile App Consulting), Harlan Brewer (Select Engineering Services), Marita Ellixson (Eglin AFB, Universidad Indiana Wesleyan, Facilitador en Jefe), John Goodson (DataDiret), Anne Horton (Lockheed Martin), Terrell Regis Hull (Logicalis Integration Solutions), Clark Richey (RABA Technologies, LLC, Java Sun Champion), Manfred Riem (UTA Interactive, LLC, Java Sun Champion), Karen Tegtmeyer (Model Technologies, Inc.), David Wolf (Universidad Pacific Lutheran) y Hua Yan (Borough of Manhattan Community Collage, City University of New York). **Revisores de la post-publicación de Cómo programar en Java 6ª edición:** Anne Horton (Lockheed Martin), William Martz (Universidad de Colorado, en Colorado Springs), Bill O'Farrell (IBM), Jeffry Babb (Universidad Virginia Commonwealth), Jeffrey Six (Universidad de Delaware, Instalaciones Adjuntas), Jesse Glick (Sun Microsystems), Karen Tegtmeyer (Model Technologies, Inc.), Kyle Gabhart (L-3 Communications), Marita Ellixson (Eglin AFB, Universidad Indiana Wesleyan, Facilitador en Jefe) y Sean Santry (Consultor independiente).

Revisores de Cómo programar en Java 6ª edición (incluyendo a los revisores de la post-publicación de la 5ª edición)

Revisores académicos: Karen Arlien (Colegio Estatal de Bismarck), Ben Blake (Universidad Estatal de Cleveland), Walt Bunch (Universidad Chapman), Marita Ellixson (Eglin AFB/Universidad de Arkansas), Ephrem Eyob (Universidad Estatal de Virginia), Bjorn Foss (Universidad Metropolitana de Florida), Bill Freitas (The Lawrenceville School), Joe Kasprzyk (Colegio Estatal de Salem), Brian Larson (Modesto Junior College), Roberto Lopez-Herrejon (Universidad de Texas en Austin), Dean Mellas (Cerritos College), David Messier (Eastern University), Andy Novobilski (Universidad de Tennessee, Chattanooga), Richard Ord (Universidad de California, San Diego), Gavin Osborne (Saskatchewan Institute of Applied Science & Technology), Donna Reese (Universidad Estatal de Mississippi), Craig Slinkman (Universidad de Texas en Arlington), Sreedhar Thota (Western Iowa Tech Community Collage), Mahendran Velauthapillai (Universidad de Georgetown), Loran Walter (Universidad Tecnológica de Lawrence) y Stephen Weiss (Universidad de Carolina del Norte en Chapel Hill). **Revisores de la industria:** Butch Anton (Wi-Tech Consulting), Jonathan Bruce (Sun Microsystems, Inc.; Líder de Especificaciones de JCP para JDBC), Gilad Bracha (Sun Microsystems, Inc.; Líder de Especificaciones de JCP para Genéricos), Michael Develle (Consultor independiente), Jonathan Gadzik (Consultor independiente), Brian Goetz (Quiotix Corporation (Miembro del Grupo de Expertos de Especificaciones de Herramientas de Concurrencia de JCP), Anne Horton (AT&T Bell Laboratories), James Huddleston (Consultor independiente), Peter Jones (Sun Microsystems, Inc.), Doug Kohlert (Sun Microsystems, Inc.), Earl LaBatt (Altaworks Corp./Universidad de New Hampshire), Paul Monday (Sun Microsystems, Inc.), Bill O'Farrell (IBM), Cameron Skinner (Embarcadero Technologies, Inc.), Brandon Taylor (Sun Microsystems, Inc.) y Karen Tegtmeyer (Consultor indepen-

diente). **Revisores del ejemplo práctico opcional de DOO/UML:** Sinan Si Alhir (Consultor independiente), Gene Ames (Star HRG), Jan Bergandy (Universidad de Massachussetts en Dartmouth), Marita Ellixson (Eglin AFB/Universidad de Arkansas), Jonathan Gadzik (Consultor independiente), Thomas Harder (ITT ESI, Inc.), James Huddleston (Consultor independiente), Terrell Hull (Consultor independiente), Kenneth Hussey (IBM), Joe Kasprzyk (Colegio Estatal de Salem), Dan McCracken (City College of New York), Paul Monday (Sun Microsystems, Inc.), Davyd Norris (Rational Software), Cameron Skinner (Embarcadero Technologies, Inc.), Craig Slinkman (Universidad de Texas en Arlington) y Steve Tockey (Construx Software).

Estos profesionales revisaron cada aspecto del libro y realizaron innumerables sugerencias para mejorar la precisión e integridad de la presentación.

Bueno ¡ahí lo tiene! Java es un poderoso lenguaje de programación que le ayudará a escribir programas con rapidez y eficiencia. Escala sin problemas hacia el ámbito del desarrollo de sistemas empresariales, para ayudar a las organizaciones a crear sus sistemas de información críticos. A medida que lea el libro, apreciaremos con sinceridad sus comentarios, críticas, correcciones y sugerencias para mejorar el texto. Dirija toda su correspondencia a:

```
deitel@deitel.com
```

Le responderemos oportunamente y publicaremos las correcciones y aclaraciones en nuestro sitio Web,

```
www.deitel.com/books/jHTP7/
```

¡Esperamos que disfrute aprendiendo con este libro tanto como nosotros disfrutamos el escribirlo!

Paul J. Deitel
Dr. Harvey M. Deitel

Maynard, Massachussets
Diciembre del 2006

Acerca de los autores

Paul J. Deitel, CEO y Director Técnico de Deitel & Associates, Inc., es egresado del Sloan School of Management del MIT (Massachussets Institute of Technology), en donde estudió Tecnología de la Información. Posee las certificaciones Programador Certificado en Java (Java Certified Programmer) y Desarrollador Certificado en Java (Java Certified Developer), y ha sido designado por Sun Microsystems como Java Champion. A través de Deitel & Associates, Inc., ha impartido cursos en Java, C, C++, C# y Visual Basic a clientes de la industria, incluyendo: IBM, Sun Microsystems, Dell, Lucent Technologies, Fidelity, NASA en el Centro Espacial Kennedy, el National Severe Storm Laboratory, White Sands Missile Range, Rogue Wave Software, Boeing, Stratus, Cambridge Technology Partners, Open Environment Corporation, One Wave, Hyperion Software, Adra Systems, Entergy, CableData Systems, Nortel Networks, Puma, iRobot, Invensys y muchos más. También ha ofrecido conferencias de Java y C++ para la *Boston Chapter of the Association for Computing Machinery*. Él y su padre, el Dr. Harvey M. Deitel, son autores de los libros de programación más vendidos en el mundo.

Dr. Harvey M. Deitel, es Presidente y Consejero de Estrategia de Deitel & Associates, Inc., tiene 45 años de experiencia en el campo de la computación; lo que incluye un amplio trabajo académico y en la industria. El Dr. Deitel tiene una licenciatura y una maestría por el MIT y un doctorado de la Universidad de Boston. Tiene 20 años de experiencia como profesor universitario, la cual incluye un puesto vitalicio y el haber sido presidente del departamento de Ciencias de la computación en el Boston College antes de fundar, con su hijo Paul J. Deitel, Deitel & Associates, Inc. Él y Paul son coautores de varias docenas de libros y paquetes multimedia, y piensan escribir muchos más. Los textos de los Deitel se han ganado el reconocimiento internacional y han sido traducidos al japonés, alemán, ruso, español, chino tradicional, chino simplificado, coreano, francés, polaco, italiano, portugués, griego, urdú y turco. El Dr. Deitel ha impartido cientos de seminarios profesionales para grandes empresas, instituciones académicas, organizaciones gubernamentales y diversos sectores del ejército.

Acerca de Deitel & Associates, Inc.

Deitel & Associates, Inc. es una empresa reconocida a nivel mundial, dedicada al entrenamiento corporativo y la creación de contenido, con especialización en lenguajes de programación, tecnología de software para Internet/World Wide Web, educación de tecnología de objetos y desarrollo de negocios por Internet a través de su

Iniciativa de Negocios en Internet. La empresa proporciona cursos, que son impartidos por instructores, sobre la mayoría de los lenguajes y plataformas de programación, como Java, Java Avanzado, C, C++, C#, Visual C++, Visual Basic, XML, Perl, Python, tecnología de objetos y programación en Internet y World Wide Web. Los fundadores de Deitel & Associates, Inc. son el Dr. Harvey M. Deitel y Paul J. Deitel. Sus clientes incluyen muchas de las empresas más grandes del mundo, agencias gubernamentales, sectores del ejército e instituciones académicas. A lo largo de su sociedad editorial de 30 años con Prentice Hall, Deitel & Associates Inc. ha publicado libros de texto de vanguardia sobre programación, libros profesionales, multimedia interactiva en CD como los *Cyber Classrooms*, *Cursos Completos de Capacitación*, cursos de capacitación basados en Web y contenido electrónico para los populares sistemas de administración de cursos WebCT, Blackboard y CourseCompass de Pearson. Deitel & Associates, Inc. y los autores pueden ser contactados mediante correo electrónico en:

```
deitel@deitel.com
```

Para conocer más acerca de Deitel & Associates, Inc., sus publicaciones y su currículum mundial de la Serie de Capacitación Corporativa *DIVE INTO*®, visite:

```
www.deitel.com
```

y suscríbase al boletín gratuito de correo electrónico, *Deitel® Buzz Online*, en:

```
www.deitel.com/newsletter/subscribe.html
```

Puede verificar la lista creciente de Centros de Recursos Deitel en:

```
www.deitel.com/resourcecenters.html
```

Quienes deseen comprar publicaciones de Deitel pueden hacerlo en:

```
www.deitel.com/books/index.html
```

Las empresas, el gobierno, las instituciones militares y académicas que deseen realizar pedidos en masa deben hacerlo directamente con Prentice Hall. Para obtener más información, visite:

```
www.prenhall.com/mischtm/support.html#order
```

Antes de empezar

Antes de comenzar a utilizar este libro, debe seguir las instrucciones de esta sección para asegurarse que Java esté instalado de manera apropiada en su computadora.

Convenciones de fuentes y nomenclatura

Utilizamos varios tipos de letra para diferenciar los componentes en la pantalla (como los nombres de menús y los elementos de los mismos) y el código o los comandos en Java. Nuestra convención es hacer hincapié en los componentes en pantalla en una fuente **Helvetica** sans-serif en negritas (por ejemplo, el menú **Archivo**) y enfatizar el código y los comandos de Java en una fuente `Lucida` sans-serif (por ejemplo, `System.out.println()`).

Kit de desarrollo de Java Standard Edition (JDK) 6

Los ejemplos en este libro se desarrollaron con el Kit de Desarrollo de Java Standard Edition (JDK) 6. Puede descargar la versión más reciente y su documentación en:

```
java.sun.com/javase/6/download.jsp
```

Si tiene preguntas, envíe un correo electrónico a `deitel@deitel.com`. Le responderemos en breve.

Requerimientos de software y hardware del sistema

- Procesador Pentium III de 500 MHz (mínimo) o de mayor velocidad; Sun® Java™ Studio Creator 2 Update 1 requiere un procesador Intel Pentium 4 de 1 GHz (o equivalente).
- Microsoft Windows Server 2003, Windows XP (con Service Pack 2), Windows 2000 Professional (con Service Pack 4).
- Una de las siguientes distribuciones de Linux: Red Hat® Enterprise Linux 3, o Red Hat Fedora Core 3.
- Mínimo 512 MB de memoria en RAM; Sun Java Studio Creator 2 Update 1 requiere 1 GB de RAM.
- Mínimo 1.5 GB de espacio en disco duro.
- Unidad de CD-ROM.
- Conexión a Internet.
- Explorador Web, Adobe® Acrobat® Reader® y una herramienta para descomprimir archivos zip.

Uso de los CD

Los ejemplos para *Cómo programar en Java, 7ª edición* se encuentran en los CD (Windows y Linux) que se incluyen en este libro. Siga los pasos de la siguiente sección, *Cómo copiar los ejemplos del libro del CD*, para copiar el directorio de ejemplos apropiado del CD a su disco duro. Le sugerimos trabajar desde su disco duro en lugar de hacerlo desde su unidad de CD por dos razones: 1, los CD son de sólo lectura, por lo que no podrá guardar sus aplicaciones en ellos; 2 es posible acceder a los archivos con mayor rapidez desde un disco duro que de un CD. Los ejemplos del libro también están disponibles para descargarse de:

```
www.deitel.com/books/jhtp7/
www.pearsoneducacion.net/deitel/
```

La interfaz para el contenido del CD de Microsoft® Windows® está diseñada para iniciarse de manera automática, a través del archivo AUTORUN.EXE. Si no aparece una pantalla de inicio cuando inserte el CD en su

computadora, haga doble clic en el archivo `welcome.htm` para iniciar la interfaz del CD para el estudiante, o consulte el archivo `readme.txt` en el CD. Para iniciar la interfaz del CD para Linux, haga doble clic en el archivo `welcome.html`.

Cómo copiar los ejemplos del libro del CD

Las capturas de pantalla de esta sección pueden diferir un poco de lo que usted verá en su computadora, de acuerdo con el sistema operativo y el explorador Web de que disponga. Las instrucciones de los siguientes pasos asumen que está utilizando Microsoft Windows.

1. ***Insertar el CD.*** Inserte el CD que se incluye con este libro en la unidad de CD de su computadora. A continuación deberá aparecer de manera automática la página Web `welcome.htm` (figura 1) en Windows. También puede utilizar el Explorador de Windows para ver el contenido del CD y hacer doble clic en `welcome.htm` para mostrar esta página.
2. ***Abrir el directorio del CD-ROM.*** Haga clic en el vínculo **Browse CD Contents** (Explorar contenido del CD) (figura 1) para ver el contenido del CD.

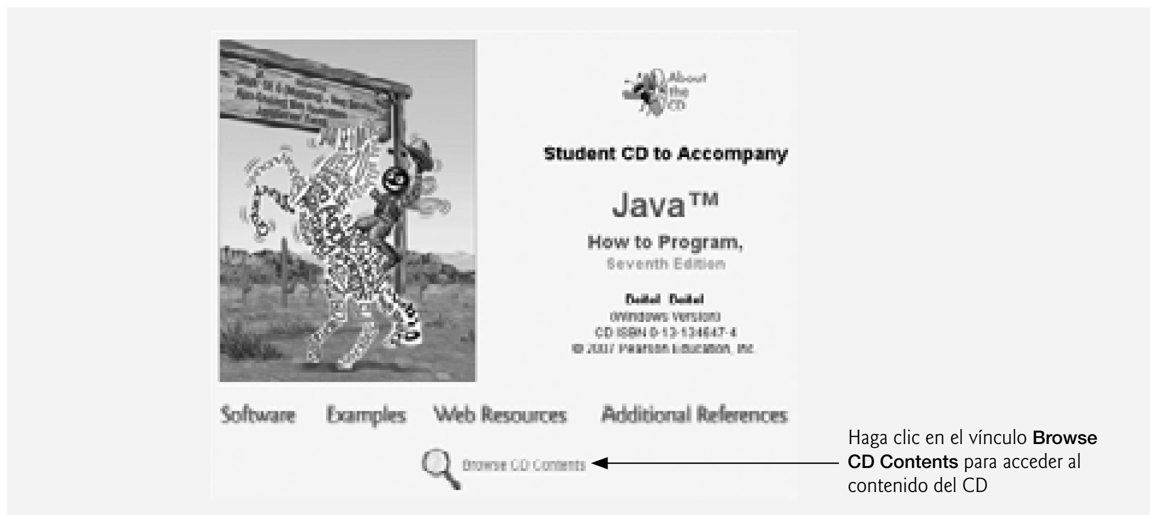


Figura 1 | Página de bienvenida para el CD de *Cómo programar en Java.*

3. ***Copiar el directorio* ejemplos.** Haga clic en el directorio `ejemplos` (figura 2), después seleccione **Copiar**. A continuación, use el Explorador de Windows para ver el contenido de su unidad **C:**. (Tal vez necesite hacer clic en un vínculo para mostrar el contenido de la unidad). Una vez que se muestre el contenido, haga clic en cualquier parte y seleccione la opción **Pegar** del menú **Editar** para copiar el directorio `ejemplos` del CD a su unidad **C:**. [*Nota:* guardamos los ejemplos directamente en la unidad **C:** y hacemos referencia a esta unidad a lo largo del texto. Puede optar por guardar sus archivos en una unidad distinta, con base en la configuración de su computadora, en el laboratorio de su escuela o sus preferencias personales. Si trabaja en un laboratorio de computadoras, consulte con su profesor para obtener más información para confirmar en dónde se deben guardar los ejemplos].

Modificación de la propiedad de sólo lectura de los archivos

Los archivos de ejemplo que copió a su computadora desde el CD son de sólo lectura. A continuación eliminará la propiedad de sólo lectura, para poder modificar y ejecutar los ejemplos.

1. ***Abrir el cuadro de diálogo* Propiedades.** Haga clic con el botón derecho del ratón en el directorio `ejemplos` y seleccione **Propiedades**. A continuación aparecerá el cuadro de diálogo **Propiedades de ejemplos** (figura 3).



Figura 2 | Copia del directorio **ejemplos**.



Figura 3 | Cuadro de diálogo **Propiedades de ejemplos**.

2. ***Cambiar la propiedad de sólo lectura.*** En la sección **Atributos** de este cuadro de diálogo, haga clic en el botón **Sólo lectura** para eliminar la marca de verificación (figura 4). Haga clic en **Aplicar** para aplicar los cambios.

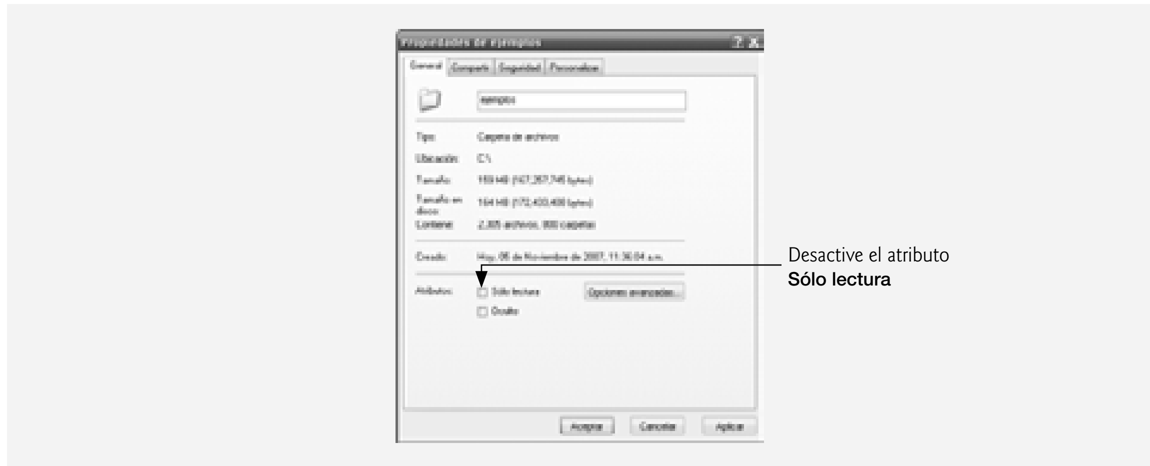


Figura 4 | Desactivar la casilla de verificación **Sólo lectura**.

3. ***Cambiar la propiedad para todos los archivos.*** Al hacer clic en **Aplicar** se mostrará la ventana **Confirmar cambios de atributos** (figura 5). En esta ventana, haga clic en el botón de opción **Aplicar cambios a esta carpeta y a todas las subcarpetas y archivos** y haga clic en **Aceptar** para eliminar la propiedad de sólo lectura para todos los archivos y directorios en el directorio **ejemplos**.

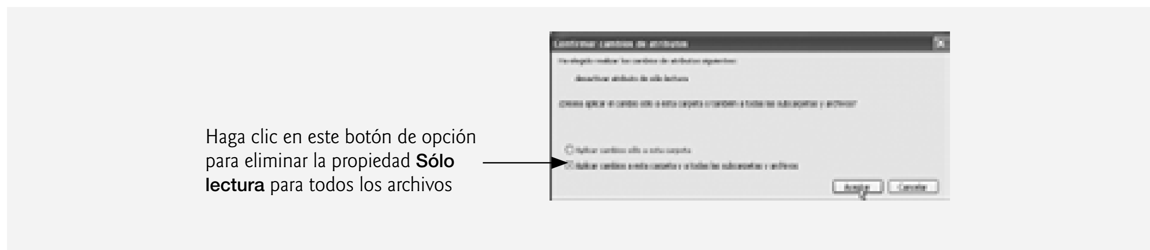


Figura 5 | Eliminar la propiedad de sólo lectura para todos los archivos en el directorio **ejemplos**.

Instalación del Kit de Desarrollo de Java Standard Edition (JDK)

Antes de ejecutar las aplicaciones de este libro o de crear sus propias aplicaciones, debe instalar el Kit de Desarrollo de Java Standard Edition (JDK) 6 o una herramienta de desarrollo para Java que soporte a Java SE 6.

Puede descargar el JDK 6 y su documentación de `java.sun.com/javase/6/download.jsp`. Haga clic en el botón **» DOWNLOAD** para JDK 6. Debe aceptar el acuerdo de licencia antes de descargar. Una vez que acepte el acuerdo, haga clic en el vínculo para el instalador de su plataforma. Guarde el instalador en su disco duro y no olvide en dónde lo guardó. Antes de instalar, lea con cuidado las instrucciones de instalación del JDK para su plataforma, que se encuentran en `java.sun.com/javase/6/webnotes/install/index.html`.

Después de descargar el instalador del JDK, haga doble clic en el programa instalador para empezar a instalarlo. Le recomendamos que acepte todas las opciones de instalación predeterminadas. Si modifica el directorio predeterminado, asegúrese de anotar el nombre y la ubicación exactos que eligió, ya que necesitará esta información más adelante en el proceso de instalación. En Windows, el JDK se coloca, de manera predeterminada, en el siguiente directorio:

```
C:\Archivos de programa\Java\jdk1.6.0
```

Establecer la variable de entorno PATH

La variable de entorno PATH en su computadora indica qué directorios debe buscar la computadora cuando intente localizar aplicaciones, como aquellas que le permiten compilar y ejecutar sus aplicaciones en Java (conocidas como `javac.exe` y `java.exe`, respectivamente). Ahora aprenderá a establecer la variable de entorno PATH en su computadora para indicar en dónde están instaladas las herramientas del JDK.

1. ***Abrir el cuadro de diálogo* Propiedades del sistema.** Haga clic en **Inicio > Panel de control > Sistema** para mostrar el cuadro de diálogo **Propiedades del sistema** (figura 6). [*Nota*: su cuadro de diálogo **Propiedades del sistema** puede tener una apariencia distinta al que se muestra en la figura 6, dependiendo de la versión de Microsoft Windows. Este cuadro de diálogo específico es de una computadora que ejecuta Microsoft Windows XP. Sin embargo, el que aparece en su computadora podría incluir distinta información].
2. ***Abrir el cuadro de diálogo* Variables de entorno.** Seleccione la ficha **Opciones avanzadas** de la parte superior del cuadro de diálogo **Propiedades del sistema** (figura 7). Haga clic en el botón **Variables de entorno** para desplegar el cuadro de diálogo **Variables de entorno** (figura 8).
3. ***Editar la variable* PATH.** Desplácese por el cuadro **Variables del sistema** para seleccionar la variable PATH. Haga clic en el botón **Modificar**. Esto hará que se despliegue el cuadro de diálogo **Modificar la variable del sistema** (figura 9).
4. ***Modificar la variable* PATH.** Coloque el cursor dentro del campo **Valor de variable**. Use la flecha izquierda para desplazar el cursor hasta el inicio de la lista. Al principio de la lista, escriba el nombre del directorio en el que colocó el JDK, seguido de `\bin;` (figura 10). Agregue `C:\Archivos de programa \Java\jdk1.6.0\bin;` a la variable PATH, si eligió el directorio de instalación predeterminado. *No coloque espacios antes o después de lo que escriba*. No se permiten espacios antes o después de cada valor en una variable de entorno. Haga clic en el botón **Aceptar** para aplicar sus cambios a la variable PATH.

Si no establece la variable PATH de manera correcta, al utilizar las herramientas del JDK recibirá un mensaje como éste:

```
'java' no se reconoce como un comando interno o externo,
programa o archivo por lotes ejecutable.
```

En este caso, regrese al principio de esta sección y vuelva a comprobar sus pasos. Si ha descargado una versión más reciente del JDK, tal vez necesite modificar el nombre del directorio de instalación del JDK en la variable PATH.



Figura 6 | Cuadro de diálogo **Propiedades del sistema**.

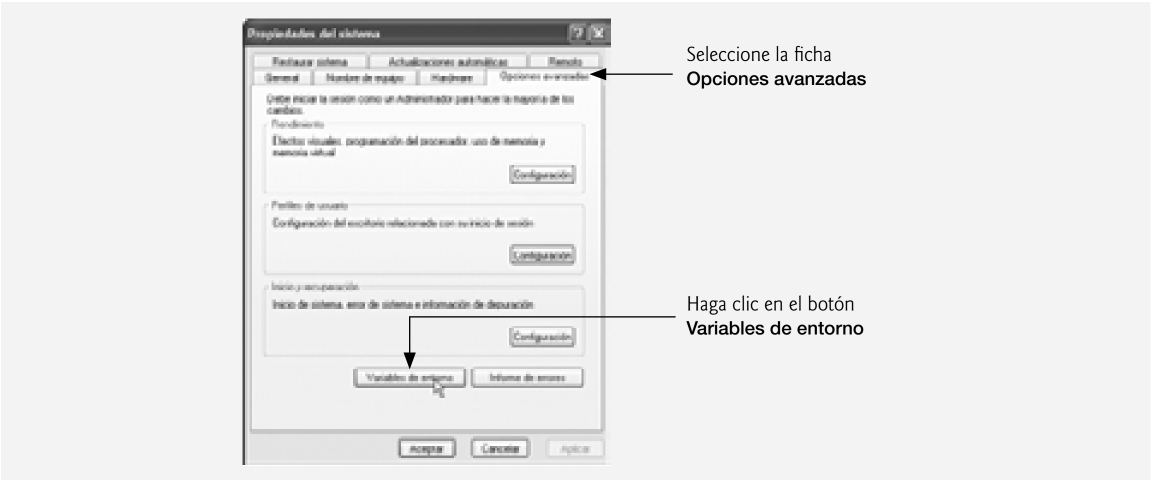


Figura 7 | Ficha **Opciones avanzadas** del cuadro de diálogo **Propiedades del sistema**.



Figura 8 | Cuadro de diálogo **Variables de entorno**.

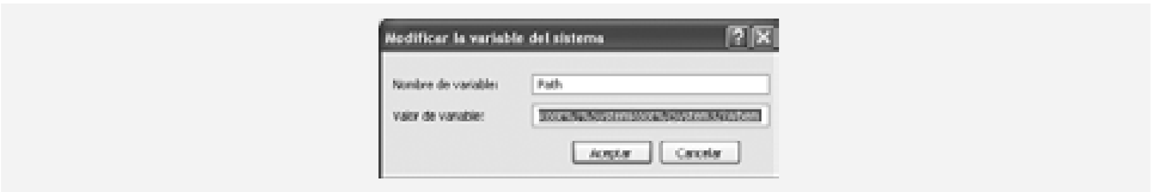


Figura 9 | Cuadro de diálogo **Modificar la variable del sistema**.

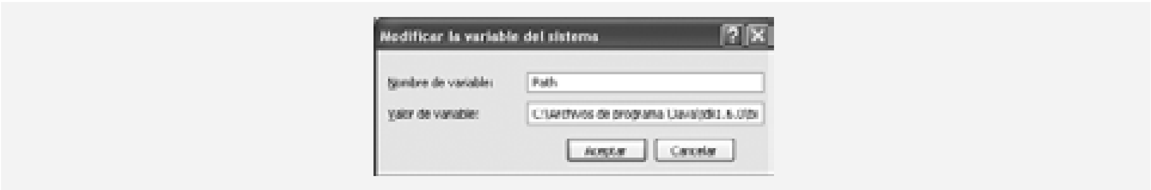


Figura 10 | Modificación de la variable PATH.

Establecer la variable de entorno CLASSPATH

Si trata de ejecutar un programa en Java y recibe un mensaje como:

```
Exception in thread "main" java.lang.NoClassDefFoundError: SuClase
```

entonces su sistema tiene una variable de entorno CLASSPATH que debe modificarse. Para corregir el error anterior, siga los pasos para establecer la variable de entorno PATH, localice la variable CLASSPATH y modifique su valor para que incluya lo siguiente:

```
.;
```

al principio de su valor (sin espacios antes o después de estos caracteres).

Ahora está listo para empezar sus estudios de Java con el libro *Cómo programar en Java, 7ª edición*. ¡Esperamos que lo disfrute!

1

Introducción a las computadoras, Internet y Web



Nuestra vida se malgasta por los detalles… simplificar, simplificar.

—Henry David Thoreau

La principal cualidad del lenguaje es la claridad.

—Galen

Mi sublime objetivo deberé llevarlo a cabo a tiempo.

—W. S. Gilbert

Tenía un maravilloso talento para empacar estrechamente el pensamiento, haciéndolo portable.

—Thomas B. Macaulay

"¡Caray, creo que de los dos, el intérprete es el más difícil de entender!"

—Richard Brinsley Sheridan

El hombre sigue siendo la computadora más extraordinaria de todas.

—John F. Kennedy

OBJETIVOS

En este capítulo aprenderá a:

- Comprender los conceptos básicos de hardware y software.
- Conocer los conceptos básicos de la tecnología de objetos, como las clases, objetos, atributos, comportamientos, encapsulamiento, herencia y polimorfismo.
- Familiarizarse con los distintos lenguajes de programación.
- Saber qué lenguajes de programación se utilizan más.
- Comprender un típico entorno de desarrollo en Java.
- Entender el papel de Java en el desarrollo de aplicaciones cliente/servidor distribuidas para Internet y Web.
- Conocer la historia de UML: el lenguaje de diseño orientado a objetos estándar en la industria.
- Conocer la historia de Internet y World Wide Web.
- Probar aplicaciones en Java.

Plan general

1.1 Introducción

¡Bienvenido a Java! Hemos trabajado duro para crear lo que pensamos será una experiencia de aprendizaje informativa, divertida y retadora para usted. Java es un poderoso lenguaje de programación, divertido para los principiantes y apropiado para los programadores experimentados que desarrollan sistemas de información de tamaño considerable. *Cómo programar en Java, 7ª edición* es una herramienta efectiva de aprendizaje para cada una de estas audiencias.

Pedagogía

La parte central del libro se enfoca en la *claridad* de los programas, a través de las técnicas comprobadas de la *programación orientada a objetos*. Los principiantes aprenderán programación de manera correcta, desde el principio. La presentación es clara, simple y tiene muchas ilustraciones. Incluye cientos de programas completos y funcionales en Java, y muestra la salida que se obtiene al ejecutar estos programas en una computadora. Enseñamos las características de Java en un contexto de programas completos y funcionales; a esto le llamamos el **método de código activo** (Live-Code™). Los programas de ejemplo están disponibles en el CD que acompaña a este libro. También puede descargarlos de los sitios Web `www.deitel.com/books/jhtp7/` o `www.pearsoneducacion.net.com/deitel`.

Fundamentos

Los primeros capítulos presentan los fundamentos de las computadoras, la programación de éstas y el lenguaje de programación Java, con lo cual se provee una base sólida para un análisis más detallado de Java en los capítulos posteriores. Los programadores experimentados tienden a leer los primeros capítulos rápidamente, y descubren que el análisis de Java en los capítulos posteriores es riguroso y retador.

La mayoría de las personas están familiarizadas con las emocionantes tareas que realizan las computadoras. Por medio de este libro, usted aprenderá a programar las computadoras para que realicen dichas tareas. El

software (las instrucciones que usted escribe para indicar a la computadora que realice **acciones** y tome **decisiones**) es quien controla a las computadoras (conocidas comúnmente como hardware). Java, desarrollado por Sun Microsystems, es uno de los lenguajes para desarrollo de software más populares en la actualidad.

Java Standard Edition 6 (Java SE 6) y el Kit de Desarrollo de Java 6 (JDK 6)

Este libro se basa en la plataforma **Java Standard Edition 6 (Java SE 6)** de Sun, también conocida como Mustang. Sun ofrece una implementación de Java SE 6, conocida como **Kit de Desarrollo de Java (JDK)**, que incluye las herramientas necesarias para escribir software en Java. Nosotros utilizamos el JDK versión 6.0 para los programas en este libro. Por lo regular, Sun actualiza el JDK para corregir errores: para descargar la versión más reciente del JDK 6, visite `java.sun.com/javase/6/download.jsp`.

Evolución de la computación y de la programación

El uso de las computadoras se está incrementando en casi cualquier campo de trabajo; los costos de se han reducido en forma dramática, debido al rápido desarrollo en la tecnología de hardware y software. Las computadoras que ocupaban grandes habitaciones y que costaban millones de dólares, hace algunas décadas, ahora pueden colocarse en las superficies de chips de silicio más pequeños que una uña, y con un costo de quizá unos cuantos dólares cada uno. Por fortuna, el silicio es uno de los materiales más abundantes en el planeta (es uno de los ingredientes de la tierra). La tecnología de los chips de silicio ha vuelto tan económica a la tecnología de la computación que cientos de millones de computadoras de uso general se encuentran actualmente ayudando a la gente de todo el mundo en: empresas, la industria, el gobierno y en sus vidas. Dicho número podría duplicarse fácilmente en unos cuantos años.

A través de los años, muchos programadores aprendieron la metodología conocida como programación estructurada. Usted aprenderá tanto la programación estructurada como la novedosa y excitante metodología de la **programación orientada a objetos**. ¿Por qué enseñamos ambas? La programación orientada a objetos es la metodología clave utilizada hoy en día por los programadores. Usted creará y trabajará con muchos objetos de software en este libro. Sin embargo, descubrirá que la estructura interna de estos objetos se construye, a menudo, utilizando técnicas de programación estructurada. Además, la lógica requerida para manipular objetos se expresa algunas veces mediante la programación estructurada.

El lenguaje de elección para las aplicaciones en red

Java se ha convertido en el lenguaje de elección para implementar aplicaciones basadas en Internet, y software para dispositivos que se comunican a través de una red. Ahora, los estéreos y otros dispositivos en los hogares pueden conectarse entre sí mediante el uso de tecnología Java. ¡En la conferencia JavaOne en mayo del 2006, Sun anunció que había mil millones de teléfonos móviles y dispositivos portátiles habilitados para Java! Java ha evolucionado rápidamente en el ámbito de las aplicaciones de gran escala. Es el lenguaje preferido para satisfacer la mayoría de las necesidades de programación de muchas organizaciones.

Java ha evolucionado tan rápidamente que publicamos esta séptima edición de *Cómo programar en Java* justamente 10 años después de publicar la primera edición. Java ha crecido tanto que cuenta con otras dos ediciones. La edición **Java Enterprise Edition (Java EE)** está orientada hacia el desarrollo de aplicaciones de red distribuidas, de gran escala, y aplicaciones basadas en Web. La plataforma **Java Micro Edition (Java ME)** está orientada hacia el desarrollo de aplicaciones para dispositivos pequeños, con memoria limitada, como los teléfonos celulares, radiolocalizadores y PDAs.

Permanezca en contacto con nosotros

Está a punto de comenzar una ruta de desafíos y recompensas. Mientras tanto, si desea comunicarse con nosotros, envíenos un correo a `deitel@deitel.com` o explore nuestro sitio Web en `www.deitel.com`. Le responderemos a la brevedad. Para mantenerse al tanto de los desarrollos con Java en Deitel & Associates, regístrese para recibir nuestro boletín de correo electrónico, *Deitel® Buzz Online* en

```
www.deitel.com/newsletter/subscribe.html
```

Para obtener material adicional sobre Java, visite nuestra creciente lista de centros de recursos en `www.deitel.com/ResourceCenters.html`. Esperamos que disfrute aprender con *Cómo programar en Java, 7ª edición*.

1.2 ¿Qué es una computadora?

Una **computadora** es un dispositivo capaz de realizar cálculos y tomar decisiones lógicas a velocidades de millones (incluso de miles de millones) de veces más rápidas que los humanos. Por ejemplo, muchas de las computadoras personales actuales pueden realizar varios miles de millones de cálculos en un segundo. Una persona con una calculadora podría requerir toda una vida para completar el mismo número de operaciones. (Puntos a considerar: ¿cómo sabría que la persona sumó los números de manera correcta?, ¿cómo sabría que la computadora sumó los números de manera correcta?) ¡Las **supercomputadoras** actuales más rápidas pueden realizar *billones* de sumas por segundo!

Las computadoras procesan los **datos** bajo el control de conjuntos de instrucciones llamadas **programas de cómputo**. Estos programas guían a la computadora a través de conjuntos ordenados de acciones especificadas por gente conocida como **programadores de computadoras**.

Una computadora está compuesta por diversos dispositivos (como teclado, monitor, ratón, discos, memoria, DVD, CD-ROM y unidades de procesamiento) conocidos como **hardware**. A los programas que se ejecutan en una computadora se les denomina **software**. Los costos de las piezas de hardware han disminuido de manera espectacular en años recientes, al punto en que las computadoras personales se han convertido en artículos domésticos. En este libro aprenderá métodos comprobados que *pueden* reducir los costos de desarrollo del software: programación orientada a objetos y (en nuestro Ejemplo práctico de Ingeniería de Software en los capítulos 2-8 y 10) diseño orientado a objetos.

1.3 Organización de una computadora

Independientemente de las diferencias en su apariencia física, casi todas las computadoras pueden representarse mediante seis **unidades lógicas** o secciones:

1. ***Unidad de entrada.*** Esta sección "receptora" obtiene información (datos y programas de cómputo) desde diversos **dispositivos de entrada** y pone esta información a disposición de las otras unidades para que pueda procesarse. La mayoría de la información se introduce a través de los teclados y ratones; también puede introducirse de muchas otras formas, como hablar con su computadora, digitalizar imágenes y desde una red, como Internet.

2. ***Unidad de salida.*** Esta sección de "embarque" toma información que ya ha sido procesada por la computadora y la coloca en los diferentes **dispositivos de salida**, para que esté disponible fuera de la computadora. Hoy en día, la mayoría de la información de salida de las computadoras se despliega en el monitor, se imprime en papel o se utiliza para controlar otros dispositivos. Las computadoras también pueden dar salida a su información a través de redes como Internet.

3. ***Unidad de memoria.*** Esta sección de "almacén" de acceso rápido, pero con relativa baja capacidad, retiene la información que se introduce a través de la unidad de entrada, para que esté disponible de manera inmediata para procesarla cuando sea necesario. La unidad de memoria también retiene la información procesada hasta que ésta pueda colocarse en los dispositivos de salida por la unidad de salida. Por lo general, la información en la unidad de memoria se pierde cuando se apaga la computadora. Con frecuencia, a esta unidad de memoria se le llama **memoria** o **memoria primaria**.

4. ***Unidad aritmética y lógica (ALU).*** Esta sección de "manufactura" es la responsable de realizar cálculos como suma, resta, multiplicación y división. Contiene los mecanismos de decisión que permiten a la computadora hacer cosas como, por ejemplo, comparar dos elementos de la unidad de memoria para determinar si son iguales o no.

5. ***Unidad central de procesamiento (CPU).*** Esta sección "administrativa" coordina y supervisa la operación de las demás secciones. La CPU le indica a la unidad de entrada cuándo debe grabarse la información dentro de la de memoria; a la ALU, cuándo debe utilizarse la información de la memoria para los cálculos; y a la unidad de salida, cuándo enviar la información desde la memoria hasta ciertos dispositivos de salida. Muchas de las computadoras actuales contienen múltiples CPUs y, por lo tanto, pueden realizar diversas operaciones de manera simultánea (a estas computadoras se les conoce como **multiprocesadores**).

6. ***Unidad de almacenamiento secundario.*** Ésta es la sección de "almacén" de alta capacidad y de larga duración. Los programas o datos que no se encuentran en ejecución por las otras unidades, normalmente se colocan en dispositivos de almacenamiento secundario (por ejemplo, el disco duro) hasta que son requeridos nuevamente, posiblemente horas, días, meses o incluso años después. El tiempo para acceder a la información en almacenamiento secundario es mucho mayor que el que se necesita para acceder a la información de la memoria principal, pero el costo por unidad de memoria secundaria es mucho menor que el correspondiente a la unidad de memoria principal. Los CDs y DVDs son ejemplos de dispositivos de almacenamiento secundario y pueden contener hasta cientos de millones y miles de millones de caracteres, respectivamente.

1.4 Los primeros sistemas operativos

Las primeras computadoras eran capaces de realizar solamente una **tarea** o **trabajo** a la vez. A esta forma de operación de la computadora a menudo se le conoce como **procesamiento por lotes** (batch) de un solo usuario. La computadora ejecuta un solo programa a la vez, mientras procesa los datos en grupos o **lotes**. En estos primeros sistemas, los usuarios generalmente asignaban sus trabajos a un centro de cómputo que los introducía en paquetes de tarjetas perforadas, y a menudo tenían que esperar horas, o incluso días, antes de que sus resultados impresos regresaran a sus escritorios.

El software denominado **sistema operativo** se desarrolló para facilitar el uso de la computadora. Los primeros sistemas operativos administraban la suave transición entre trabajos, incrementando la cantidad de trabajo, o el **flujo de datos**, que las computadoras podían procesar.

Conforme las computadoras se volvieron más poderosas, se hizo evidente que un proceso por lotes para un solo usuario era ineficiente, debido al tiempo que se malgastaba esperando a que los lentos dispositivos de entrada/salida completaran sus tareas. Se pensó que era posible realizar muchos trabajos o tareas que podrían *compartir* los recursos de la computadora y lograr un uso más eficiente. A esto se le conoce como **multiprogramación**; que significa la operación simultánea de muchas tareas que compiten para compartir los recursos de la computadora. Aun con los primeros sistemas operativos con multiprogramación, los usuarios seguían enviando sus tareas en paquetes de tarjetas perforadas y esperaban horas, incluso hasta días, por los resultados.

En la década de los sesenta, varios grupos en la industria y en las universidades marcaron la pauta de los sistemas operativos de **tiempo compartido**. El tiempo compartido es un caso especial de la multiprogramación, ya que los usuarios acceden a la computadora a través de terminales que, por lo general, son dispositivos compuestos por un teclado y un monitor. Puede haber docenas o incluso cientos de usuarios compartiendo la computadora al mismo tiempo. La computadora en realidad no ejecuta los procesos de todos los usuarios a la vez. Lo que hace es ejecutar una pequeña porción del trabajo de un usuario y después procede a dar servicio al siguiente usuario, con la posibilidad de proporcionar el servicio a cada usuario varias veces por segundo. Así, los programas de los usuarios *aparentemente* se ejecutan de manera simultánea. Una ventaja del tiempo compartido es que el usuario recibe respuestas casi inmediatas a las peticiones.

1.5 Computación personal, distribuida y cliente/servidor

En 1977, Apple Computer popularizó el fenómeno de la **computación personal**. Las computadoras se volvieron sumamente económicas, de manera que la gente pudo adquirirlas para su uso personal o para negocios. En 1981, IBM, el vendedor de computadoras más grande del mundo, introdujo la Computadora Personal (PC) de IBM. Con esto se legitimó rápidamente la computación en las empresas, en la industria y en las organizaciones gubernamentales.

Estas computadoras eran unidades "independientes" (la gente transportaba sus discos de un lado a otro para compartir información; a esto se le conoce comúnmente como "sneakernet"). Aunque las primeras computadoras personales no eran lo suficientemente poderosas para compartir el tiempo entre varios usuarios, podían interconectarse mediante redes computacionales, algunas veces a través de líneas telefónicas y otras mediante **redes de área local (LANs)** dentro de una empresa. Esto derivó en el fenómeno denominado **computación distribuida**, en donde todos los cálculos informáticos de una empresa, en vez de realizarse estrictamente dentro de un centro de cómputo, se distribuyen mediante redes a los sitios en donde se realiza el trabajo de la empresa. Las computadoras personales eran lo suficientemente poderosas para manejar los requerimientos de cómputo de usuarios individuales, y para manejar las tareas básicas de comunicación que involucraban la transferencia de información entre una computadora y otra, de manera electrónica.

Las computadoras personales actuales son tan poderosas como las máquinas de un millón de dólares de hace apenas unas décadas. Las máquinas de escritorio más poderosas (denominadas **estaciones de trabajo**) proporcionan a cada usuario enormes capacidades. La información se comparte fácilmente a través de redes de computadoras, en donde algunas computadoras denominadas **servidores** almacenan datos que pueden ser utilizados por computadoras **cliente** distribuidas en toda la red, de ahí el término de **computación cliente/servidor**. Java se está utilizando ampliamente para escribir software para redes de computadoras y para aplicaciones cliente/servidor distribuidas. Los sistemas operativos actuales más populares como Linux, Mac OS X y Microsoft Windows proporcionan el tipo de capacidades que explicamos en esta sección.

1.6 Internet y World Wide Web

Internet (una red global de computadoras) tiene sus raíces en la década de 1960; su patrocinio estuvo a cargo del Departamento de Defensa de los Estados Unidos. Diseñada originalmente para conectar los sistemas de cómputo principales de aproximadamente una docena de universidades y organizaciones de investigación, actualmente, Internet es utilizada por cientos de millones de computadoras y dispositivos controlados por computadora en todo el mundo.

Con la introducción de **World Wide Web** (que permite a los usuarios de computadora localizar y ver documentos basados en multimedia, sobre casi cualquier tema, a través del ciberespacio), Internet se ha convertido explosivamente en uno de los principales mecanismos de comunicación en todo el mundo.

Internet y World Wide Web se encuentran, sin duda, entre las creaciones más importantes y profundas de la humanidad. En el pasado, la mayoría de las aplicaciones de computadora se ejecutaban en equipos que no estaban conectados entre sí. Las aplicaciones de la actualidad pueden diseñarse para intercomunicarse entre computadoras en todo el mundo. Internet mezcla las tecnologías de la computación y las comunicaciones. Facilita nuestro trabajo. Hace que la información esté accesible en forma instantánea y conveniente para todo el mundo. Hace posible que los individuos y negocios pequeños locales obtengan una exposición mundial. Está cambiando la forma en que se hacen los negocios. La gente puede buscar los mejores precios para casi cualquier producto o servicio. Los miembros de las comunidades con intereses especiales pueden mantenerse en contacto unos con otros. Los investigadores pueden estar inmediatamente al tanto de los últimos descubrimientos.

Cómo programar en Java, 7ª edición presenta técnicas de programación que permiten a las aplicaciones en Java utilizar Internet y Web para interactuar con otras aplicaciones. Estas técnicas, junto con otras más, permiten a los programadores de Java desarrollar el tipo de aplicaciones distribuidas de nivel empresarial que se utilizan actualmente en la industria. Se pueden escribir aplicaciones en Java para ejecutarse en cualquier tipo de computadora, con lo cual se reduce en gran parte el tiempo y el costo de desarrollo de sistemas. Si a usted le interesa desarrollar aplicaciones que se ejecuten a través de Internet y Web, aprender Java puede ser la clave para que reciba oportunidades retadoras y remuneradoras en su profesión.

1.7 Lenguajes máquina, ensambladores y de alto nivel

Los programadores escriben instrucciones en diversos lenguajes de programación, algunos de los cuales comprende directamente la computadora, mientras que otros requieren pasos intermedios de **traducción**. En la actualidad se utilizan cientos de lenguajes de computación. Éstos se dividen en tres tipos generales:

1. Lenguajes máquina.
2. Lenguajes ensambladores.
3. Lenguajes de alto nivel.

Cualquier computadora puede entender de manera directa sólo su propio **lenguaje máquina**; que es su “lenguaje natural”, y como tal, está definido por el diseño del hardware de dicha computadora. Por lo general, los lenguajes máquina consisten en cadenas de números (que finalmente se reducen a 1s y 0s) que instruyen a las computadoras para realizar sus operaciones más elementales, una a la vez. Los lenguajes máquina son **dependientes de la máquina** (es decir, un lenguaje máquina en particular puede usarse solamente en un tipo de computadora). Dichos lenguajes son difíciles de comprender para los humanos, el siguiente ejemplo muestra uno de los primeros programas en lenguaje máquina, el cual suma el pago de las horas extras al sueldo base y almacena el resultado en el sueldo bruto:

```
+1300042774
+1400593419
+1200274027
```

La programación en lenguaje máquina era demasiado lenta y tediosa para la mayoría de los programadores. En vez de utilizar las cadenas de números que las computadoras podían entender directamente, los programadores empezaron a utilizar abreviaturas del inglés para representar las operaciones elementales. Estas abreviaturas formaron la base de los **lenguajes ensambladores**. Los **programas traductores** conocidos como **ensambladores** se desarrollaron para convertir los primeros programas en lenguaje ensamblador a lenguaje máquina, a la velocidad de la computadora. A continuación se muestra un ejemplo de un programa en lenguaje ensamblador, que también suma el pago de las horas extras al sueldo base y almacena el resultado en el sueldo bruto:

```
load   sueldobase
add    sueldoextra
store  sueldobruto
```

Aunque este código es más claro para los humanos, las computadoras no lo pueden entender sino hasta que se traduce en lenguaje máquina.

El uso de las computadoras se incrementó rápidamente con la llegada de los lenguajes ensambladores, pero los programadores aún requerían de muchas instrucciones para llevar a cabo incluso hasta las tareas más simples. Para agilizar el proceso de programación se desarrollaron los **lenguajes de alto nivel**, en donde podían escribirse instrucciones individuales para realizar tareas importantes. Los programas traductores, denominados **compiladores**, convierten, a lenguaje máquina, los programas que están en lenguaje de alto nivel. Estos últimos permiten a los programadores escribir instrucciones que son muy similares al inglés común, y contienen la notación matemática común. Un programa de nómina escrito en un lenguaje de alto nivel podría contener una instrucción como la siguiente:

```
sueldoBruto = sueldoBase + sueldoExtra
```

Obviamente, desde el punto de vista del programador, los lenguajes de alto nivel son mucho más recomendables que los lenguajes máquina o ensamblador. C, C++ y los lenguajes .NET de Microsoft (por ejemplo, Visual Basic .NET, Visual C++ .NET y C#) son algunos de los lenguajes de programación de alto nivel que más se utilizan; sin embargo, Java es *el* más utilizado.

El proceso de compilación de un programa escrito en lenguaje de alto nivel a un lenguaje máquina puede tardar un tiempo considerable en la computadora. Los programas **intérpretes** se desarrollaron para ejecutar programas en lenguaje de alto nivel directamente, aunque con más lentitud. Los intérpretes son populares en los entornos de desarrollo de programas, en los cuales se agregan nuevas características y se corrigen los errores. Una vez que se desarrolla un programa por completo, se puede producir una versión compilada para ejecutarse con la mayor eficiencia.

Actualmente se sabe que existen dos formas de traducir un programa en lenguaje de alto nivel a un formato que la computadora pueda entender: compilación e interpretación. Como veremos en la sección 1.13, Java utiliza una mezcla inteligente de estas tecnologías.

1.8 Historia de C y C++

Java evolucionó de C++, el cual evolucionó de C, que a su vez evolucionó de BCPL y B. En 1967, Martin Richards desarrolló BCPL como un lenguaje para escribir software para sistemas operativos y compiladores. Ken Thompson modeló muchas características en su lenguaje B a partir del trabajo de sus contrapartes en BCPL, y utilizó a B para crear las primeras versiones del sistema operativo UNIX, en los laboratorios Bell en 1970.

El lenguaje C evolucionó a partir de B, gracias al trabajo de Dennis Ritchie en los laboratorios Bell, y se implementó originalmente en 1972. Inicialmente, se hizo muy popular como lenguaje de desarrollo para el sistema operativo UNIX. En la actualidad, la mayoría del código para los sistemas operativos de propósito general (por ejemplo, los que se encuentran en las computadoras portátiles, de escritorio, estaciones de trabajo y pequeños servidores) se escribe en C o C++.

A principios de la década de los ochenta, Bjarne Stroustrup desarrolló una extensión de C en los laboratorios Bell: C++. Este lenguaje proporciona un conjunto de características que "pulen" al lenguaje C pero, lo más impor-

tante es que proporciona la capacidad de una *programación orientada a objetos* (que describiremos con más detalle en la sección 1.16 y en todo el libro). C++ es un lenguaje híbrido: es posible programar en un estilo parecido a C, en un estilo orientado a objetos, o en ambos.

Una revolución se está gestando en la comunidad del software. Escribir software de manera rápida, correcta y económica es aún una meta difícil de alcanzar, en una época en que la demanda de nuevo y más poderoso software se encuentra a la alza. Los *objetos*, o dicho en forma más precisa (como veremos en la sección 1.16), las clases a partir de las cuales se crean los objetos, son en esencia componentes reutilizables de software. Hay objetos de: fecha, hora, audio, automóvil, personas, etcétera; de hecho, casi cualquier sustantivo puede representarse como objeto de software en términos de **atributos** (como el nombre, color y tamaño) y **comportamientos** (como calcular, desplazarse y comunicarse). Los desarrolladores de software están descubriendo que utilizar una metodología de diseño e implementación modular y orientada a objetos puede hacer más productivos a los grupos de desarrollo de software, que mediante las populares técnicas de programación anteriores, como la programación estructurada. Los programas orientados a objetos son, a menudo, más fáciles de entender, corregir y modificar. Java es el lenguaje de programación orientada a objetos que más se utiliza en el mundo.

1.9 Historia de Java

La contribución más importante a la fecha, por parte de la revolución del microprocesador, es que hizo posible el desarrollo de las computadoras personales, que ahora suman miles de millones a nivel mundial. Las computadoras personales han tenido un profundo impacto en la vida de las personas, y en la manera en que las empresas realizan y administran su negocio.

Los microprocesadores están teniendo un profundo impacto en los dispositivos electrónicos inteligentes para uso doméstico. Al reconocer esto, Sun Microsystems patrocinó en 1991 un proyecto interno de investigación denominado Green, el cual desembocó en el desarrollo de un lenguaje basado en C++ al que su creador, James Gosling, llamó Oak debido a un roble que tenía a la vista desde su ventana en las oficinas de Sun. Posteriormente se descubrió que ya existía un lenguaje de computadora con el mismo nombre. Cuando un grupo de gente de Sun visitó una cafetería local, sugirieron el nombre Java (una variedad de café) y así se quedó.

Pero el proyecto Green tuvo algunas dificultades. El mercado para los dispositivos electrónicos inteligentes de uso doméstico no se desarrollaba tan rápido a principios de los noventa como Sun había anticipado. El proyecto corría el riesgo de cancelarse. Pero para su buena fortuna, la popularidad de World Wide Web explotó en 1993 y la gente de Sun se dio cuenta inmediatamente del potencial de Java para agregar **contenido dinámico**, como interactividad y animaciones, a las páginas Web. Esto trajo nueva vida al proyecto.

Sun anunció formalmente a Java en una importante conferencia que tuvo lugar en mayo de 1995. Java generó la atención de la comunidad de negocios debido al fenomenal interés en World Wide Web. En la actualidad, Java se utiliza para desarrollar aplicaciones empresariales a gran escala, para mejorar la funcionalidad de los servidores Web (las computadoras que proporcionan el contenido que vemos en nuestros exploradores Web), para proporcionar aplicaciones para los dispositivos domésticos (como teléfonos celulares, radiolocalizadores y asistentes digitales personales) y para muchos otros propósitos.

1.10 Bibliotecas de clases de Java

Los programas en Java constan de varias piezas llamadas **clases**. Estas clases incluyen piezas llamadas **métodos**, los cuales realizan tareas y devuelven información cuando completan esas tareas. Los programadores pueden crear cada una de las piezas que necesitan para formar programas en Java. Sin embargo, la mayoría de los programadores en Java aprovechan las ricas colecciones de clases existentes en las **bibliotecas de clases de Java**, que también se conocen como **APIs (Interfaces de programación de aplicaciones) de Java**. Por lo tanto, en realidad existen dos fundamentos para conocer el "mundo" de Java. El primero es el lenguaje Java en sí, de manera que usted pueda programar sus propias clases; el segundo son las clases incluidas en las extensas bibliotecas de clases de Java. A lo largo de este libro hablaremos sobre muchas bibliotecas de clases; que proporcionan principalmente los vendedores de compiladores, pero muchas de ellas las proporcionan vendedores de software independientes (ISVs).



Observación de ingeniería de software 1.1

Utilice un método de construcción en bloques para crear programas. Evite reinventar la rueda: use piezas existentes siempre que sea posible. Esta reutilización de software es un beneficio clave de la programación orientada a objetos.

Incluimos muchos tips como **Observaciones de ingeniería de software** a lo largo del texto para explicar los conceptos que afectan y mejoran la arquitectura y calidad de los sistemas de software. También resaltamos otras clases de tips, incluyendo las **Buenas prácticas de programación** (que le ayudarán a escribir programas más claros, comprensibles, de fácil mantenimiento, y fáciles de probar y **depurar**; es decir, eliminar errores de programación), los **Errores comunes de programación** (problemas de los que tenemos que cuidarnos y evitar), **Tips de rendimiento** (que servirán para escribir programas que se ejecuten más rápido y ocupen menos memoria), **Tips de portabilidad** (técnicas que le ayudarán a escribir programas que se ejecuten, con poca o ninguna modificación, en una variedad de computadoras; estos tips también incluyen observaciones generales acerca de cómo logra Java su alto grado de portabilidad), **Tips para prevenir errores** (que le ayudarán a eliminar errores de sus programas y, lo que es más importante, técnicas que le ayudarán a escribir programas libres de errores desde el principio) y **Observaciones de apariencia visual** (que le ayudarán a diseñar la apariencia visual de las interfaces gráficas de usuario de sus aplicaciones, además de facilitar su uso). Muchas de estas técnicas y prácticas son sólo guías. Usted deberá, sin duda, desarrollar su propio estilo de programación.



Observación de ingeniería de software 1.2

Cuando programe en Java, generalmente utilizará los siguientes bloques de construcción: clases y métodos de las bibliotecas de clases, clases y métodos creados por usted mismo, y clases y métodos creados por otros y puestos a disposición suya.

La ventaja de crear sus propias clases y métodos es que sabe exactamente cómo funcionan y puede examinar el código en Java. La desventaja es el tiempo que consumen y el esfuerzo potencialmente complejo que se requiere.



Tip de rendimiento 1.1

Utilizar las clases y métodos de las APIs de Java en vez de escribir sus propias versiones puede mejorar el rendimiento de sus programas, ya que estas clases y métodos están escritos cuidadosamente para funcionar de manera eficiente. Esta técnica también reduce el tiempo de desarrollo de los programas.



Tip de portabilidad 1.1

Utilizar las clases y métodos de las APIs de Java en vez de escribir sus propias versiones mejora la portabilidad de sus programas, ya que estas clases y métodos se incluyen en todas las implementaciones de Java.



Observación de ingeniería de software 1.3

Existen diversas bibliotecas de clases que contienen componentes reutilizables de software, y están disponibles a través de Internet y Web, muchas de ellas en forma gratuita.

Para descargar la documentación de la API de Java, visite el sitio `java.sun.com/javase/6/download.jsp` de Sun para Java.

1.11 FORTRAN, COBOL, Pascal y Ada

Se han desarrollado cientos de lenguajes de alto nivel, pero sólo unos cuantos han logrado una amplia aceptación. **Fortran (FORmula TRANslator, Traductor de fórmulas)** fue desarrollado por IBM Corporation a mediados de la década de los cincuenta para utilizarse en aplicaciones científicas y de ingeniería que requerían cálculos matemáticos complejos. En la actualidad, Fortran se utiliza ampliamente en aplicaciones de ingeniería.

COBOL (COmmon Business Oriented Language, Lenguaje común orientado a negocios) fue desarrollado a finales de la década de los cincuenta por fabricantes de computadoras, el gobierno estadounidense y usuarios de computadoras de la industria. COBOL se utiliza en aplicaciones comerciales que requieren de una manipulación precisa y eficiente de grandes volúmenes de datos. Gran parte del software de negocios aún se programa en COBOL.

Durante la década de los sesenta, muchos de los grandes esfuerzos para el desarrollo de software encontraron severas dificultades. Los itinerarios de software generalmente se retrasaban, los costos rebasaban en gran medida a los presupuestos y los productos terminados no eran confiables. La gente comenzó a darse cuenta de que el desarrollo de software era una actividad mucho más compleja de lo que habían imaginado. Las actividades de

investigación en la década de los sesenta dieron como resultado la evolución de la **programación estructurada** (un método disciplinado para escribir programas que fueran más claros, fáciles de probar y depurar, y más fáciles de modificar que los programas extensos producidos con técnicas anteriores).

Uno de los resultados más tangibles de esta investigación fue el desarrollo del lenguaje de programación Pascal por el profesor Niklaus Wirth, en 1971. Pascal, cuyo nombre se debe al matemático y filósofo Blaise Pascal del siglo diecisiete, se diseñó para la enseñanza de la programación estructurada en ambientes académicos, y de inmediato se convirtió en el lenguaje de programación preferido en la mayoría de las universidades. Pascal carece de muchas de las características necesarias para poder utilizarse en aplicaciones comerciales, industriales y gubernamentales, por lo que no ha sido muy aceptado en estos entornos.

El lenguaje de programación **Ada** se desarrolló bajo el patrocinio del Departamento de Defensa de los Estados Unidos (DOD) durante la década de los setenta y los primeros años de la década de los ochenta. Cientos de lenguajes independientes se utilizaron para producir los sistemas de software masivos de comando y control del departamento de defensa. Éste quería un solo lenguaje que pudiera satisfacer la mayoría de sus necesidades. El nombre del lenguaje es en honor de Lady Ada Lovelace, hija del poeta Lord Byron. A Lady Lovelace se le atribuye el haber escrito el primer programa para computadoras en el mundo, a principios de la década de1800 (para la Máquina Analítica, un dispositivo de cómputo mecánico diseñado por Charles Babbage). Una de las características importantes de Ada se conoce como **multitarea**, la cual permite a los programadores especificar que muchas actividades ocurran en paralelo. Java, a través de una técnica que se conoce como *subprocesamiento múltiple*, también permite a los programadores escribir programas con actividades paralelas.

1.12 BASIC, Visual Basic, Visual C++, C# y .NET

El lenguaje de programación **BASIC** (Beginner´s All-Purpose Symbolic Instruction Code, Código de instrucciones simbólicas de uso general para principiantes) fue desarrollado a mediados de la década de los sesenta en el Dartmouth College, como un medio para escribir programas simples. El propósito principal de BASIC era que los principiantes se familiarizaran con las técnicas de programación.

El lenguaje Visual Basic de Microsoft se introdujo a principios de la década de los noventa para simplificar el desarrollo de aplicaciones para Microsoft Windows, y es uno de los lenguajes de programación más populares en el mundo.

Las herramientas de desarrollo más recientes de Microsoft forman parte de su estrategia a nivel corporativo para integrar Internet y Web en las aplicaciones de computadora. Esta estrategia se implementa en la **plataforma .NET** de Microsoft, la cual proporciona a los desarrolladores las herramientas que necesitan para crear y ejecutar aplicaciones de computadora que puedan ejecutarse en computadoras distribuidas a través de Internet. Los tres principales lenguajes de programación de Microsoft son **Visual Basic .NET** (basado en el lenguaje BASIC original), **Visual C++ .NET** (basado en C++) y **C#** (basado en C++ y Java, y desarrollado expresamente para la plataforma .NET). Los desarrolladores que utilizan .NET pueden escribir componentes de software en el lenguaje con el que estén más familiarizados y formar aplicaciones al combinar esos componentes con los ya escritos en cualquier lenguaje .NET.

1.13 Entorno de desarrollo típico en Java

Ahora explicaremos los pasos típicos usados para crear y ejecutar un programa en Java, utilizando un entorno de desarrollo de Java (el cual se ilustra en la figura 1.1).

Por lo general, los programas en Java pasan a través de cinco fases: **edición**, **compilación**, **carga**, **verificación** y **ejecución**. Hablamos sobre estos conceptos en el contexto del JDK 6.0 de Sun Microsystems, Inc. Puede descargar el JDK más actualizado y su documentación en `java.sun.com/javase/6/download.jsp`. *Siga cuidadosamente las instrucciones de instalación para el JDK que se proporcionan en la sección* Antes de empezar *(o en `java.sun.com/javase/6/webnotes/install/index.htm`) para asegurarse de configurar su computadora apropiadamente para compilar y ejecutar programas en Java.* También es conveniente que visite el centro para principiantes en Java (New to Java Center) de Sun en:

```
java.sun.com/developer/onlineTraining/new2java/index.html
```

[*Nota:* este sitio Web proporciona las instrucciones de instalación para Windows, Linux y MacOS X. Si usted no utiliza uno de estos sistemas operativos, consulte los manuales del entorno de Java de su sistema, o pregunte a su

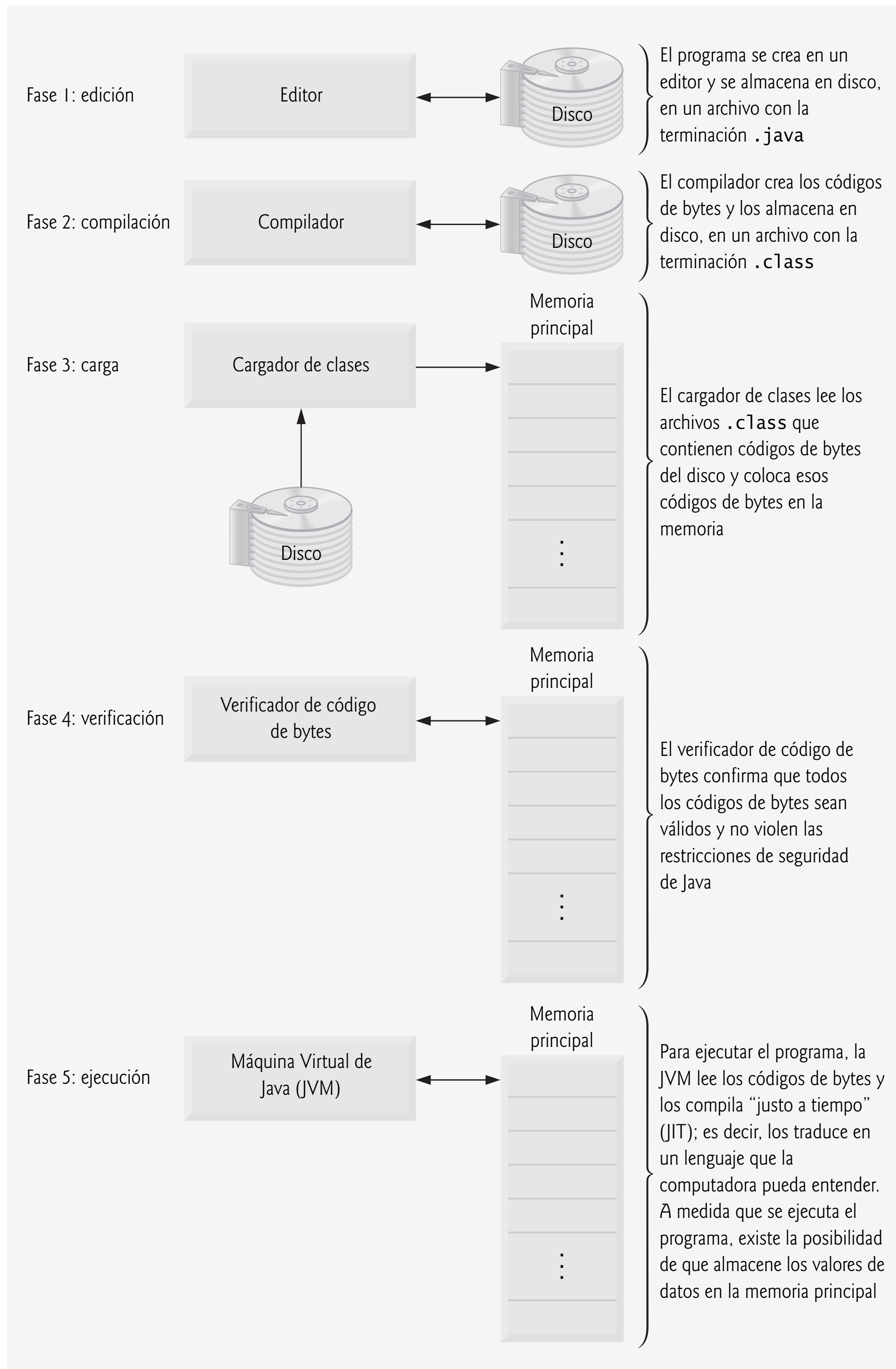


Figura 1.1 | Entorno de desarrollo típico de Java.

instructor cómo puede lograr estas tareas con base en el sistema operativo de su computadora. Además, tenga en cuenta que en ocasiones los vínculos Web se rompen a medida que las compañías evolucionan sus sitios Web. Si encuentra un problema con este vínculo o con cualquier otro al que se haga referencia en este libro, visite `www.deitel.com` para consultar la fe de erratas y notifíquenos su problema al correo electrónico `deitel@deitel.com`. Le responderemos a la brevedad].

Fase 1: Creación de un programa

La fase 1 consiste en editar un archivo con un **programa de edición** (conocido comúnmente como **editor**). Usted escribe un programa en Java (conocido, por lo general, como **código fuente**) utilizando el editor, realiza las correcciones necesarias y guarda el programa en un dispositivo de almacenamiento secundario, como su disco duro. Un nombre de archivo que termina con la **extensión `.java`** indica que éste contiene código fuente en Java. En este libro asumimos que usted ya sabe cómo editar un archivo.

Dos de los editores que se utilizan ampliamente en sistemas Linux son `vi` y `emacs`. En Windows, basta con usar un programa editor simple, como el Bloc de notas. También hay muchos editores de freeware y shareware disponibles para descargarlos de Internet, en sitios como `www.download.com`.

Para las organizaciones que desarrollan sistemas de información extensos, hay **entornos de desarrollo integrados (IDEs)** disponibles de la mayoría de los proveedores de software, incluyendo Sun Microsystems. Los IDEs proporcionan herramientas que dan soporte al proceso de desarrollo del software, incluyendo editores para escribir y editar programas, y depuradores para localizar errores lógicos.

Los IDEs populares son: Eclipse (`www.eclipse.org`), NetBeans (`www.netbeans.org`), JBuilder (`www.borland.com`), JCreator (`www.jcreator.com`), BlueJ (`www.blueJ.org`), jGRASP (`www.jgrasp.org`) y JEdit (`www.jedit.org`). Java Studio Enterprise de Sun Microsystems (`developers.sun.com/prodtech/javatools/jsenterprise/index.jsp`) es una versión mejorada de NetBeans. [*Nota:* la mayoría de nuestros programas de ejemplo deben operar de manera apropiada con cualquier entorno de desarrollo integrado de Java que cuente con soporte para el JDK 6].

Fase 2: Compilación de un programa en Java para convertirlo en códigos de bytes

En la fase 2, el programador utiliza el comando **`javac`** (el **compilador de Java**) para **compilar** un programa. Por ejemplo, para compilar un programa llamado `Bienvenido.java`, escriba

```
javac Bienvenido.java
```

en la ventana de comandos de su sistema (es decir, el **indicador de MS-DOS** en Windows 95/98/ME, el **Símbolo del sistema** en Windows NT/2000/XP, el **indicador de shell** en Linux o la **aplicación Terminal** en Mac OS X). Si el programa se compila, el compilador produce un archivo **`.class`** llamado `Bienvenido.class`, que contiene la versión compilada del programa.

El compilador de Java traduce el código fuente en **códigos de bytes** que representan las tareas a ejecutar en la fase de ejecución (fase 5). La **Máquina Virtual de Java (JVM)**, una parte del JDK y la base de la plataforma Java, ejecuta los códigos de bytes. Una **máquina virtual (VM)** es una aplicación de software que simula a una computadora, pero oculta el sistema operativo y el hardware subyacentes de los programas que interactúan con la VM. Si se implementa la misma VM en muchas plataformas computacionales, las aplicaciones que ejecute se podrán utilizar en todas esas plataformas. La JVM es una de las máquinas virtuales más utilizadas.

A diferencia del lenguaje máquina, que depende del hardware de una computadora específica, los códigos de bytes son instrucciones independientes de la plataforma; no dependen de una plataforma de hardware en especial. Entonces, los códigos de bytes de Java son **portables** (es decir, se pueden ejecutar en cualquier plataforma que contenga una JVM que comprenda la versión de Java en la que se compilaron). La JVM se invoca mediante el comando **`java`**. Por ejemplo, para ejecutar una aplicación llamada `Bienvenido`, debe escribir el comando

```
java Bienvenido
```

en una ventana de comandos para invocar la JVM, que a su vez inicia los pasos necesarios para ejecutar la aplicación. Esto comienza la fase 3.

Fase 3: Cargar un programa en memoria

En la fase 3, el programa debe colocarse en memoria antes de ejecutarse; a esto se le conoce como **cargar**. El **cargador de clases** toma los archivos `.class` que contienen los códigos de bytes del programa y los transfiere a

la memoria principal. El cargador de clases también carga cualquiera de los archivos `.class` que su programa utilice, y que sean proporcionados por Java. Puede cargar los archivos `.class` desde un disco en su sistema o a través de una red (como la de su universidad local o la red de la empresa, o incluso desde Internet).

Fase 4: Verificación del código de bytes
En la fase 4, a medida que se cargan las clases, el **verificador de códigos de bytes** examina sus códigos de bytes para asegurar que sean válidos y que no violen las restricciones de seguridad. Java implementa una estrecha seguridad para asegurar que los programas que llegan a través de la red no dañen sus archivos o su sistema (como podrían hacerlo los virus de computadora y los gusanos).

Fase 5: Ejecución
En la fase 5, la JVM ejecuta los códigos de bytes del programa, realizando así las acciones especificadas por el mismo. En las primeras versiones de Java, la JVM era tan sólo un intérprete de códigos de bytes de Java. Esto hacía que la mayoría de los programas se ejecutaran con lentitud, ya que la JVM tenía que interpretar y ejecutar un código de bytes a la vez. Por lo general, las JVMs actuales ejecutan códigos de bytes usando una combinación de la interpretación y la denominada **compilación justo a tiempo (JIT)**. En este proceso, la JVM analiza los códigos de bytes a medida que se interpretan, buscando **puntos activos**: partes de los códigos de bytes que se ejecutan con frecuencia. Para estas partes, un **compilador justo a tiempo (JIT)** (conocido como **compilador HotSpot de Java**) traduce los códigos de bytes al lenguaje máquina correspondiente a la computadora. Cuando la JVM encuentra estas partes compiladas nuevamente, se ejecuta el código en lenguaje máquina, que es más rápido. Por ende, los programas en Java en realidad pasan por dos fases de compilación: una en la cual el código fuente se traduce a código de bytes (para tener portabilidad a través de las JVMs en distintas plataformas computacionales) y otra en la que, durante la ejecución, los códigos de bytes se traducen en lenguaje máquina para la computadora actual en la que se ejecuta el programa.

Problemas que pueden ocurrir en tiempo de ejecución
Es probable que los programas no funcionen la primera vez. Cada una de las fases anteriores puede fallar, debido a diversos errores que describiremos en este texto. Por ejemplo, un programa en ejecución podría intentar una división entre cero (una operación ilegal para la aritmética con números enteros en Java). Esto haría que el programa de Java imprimiera un mensaje de error. Si esto ocurre, tendría que regresar a la fase de edición, hacer las correcciones necesarias y proseguir con las fases restantes nuevamente, para determinar que las correcciones resolvieron el(los) problema(s). [*Nota:* la mayoría de los programas en Java reciben o producen datos. Cuando decimos que un programa muestra un mensaje, por lo general, queremos decir que muestra ese mensaje en la pantalla de su computadora. Los mensajes y otros datos pueden enviarse a otros dispositivos, como los discos y las impresoras, o incluso a una red para transmitirlos a otras computadoras].



Error común de programación 1.1

*Los errores, como la división entre cero, ocurren a medida que se ejecuta un programa, de manera que a estos errores se les llama **errores en tiempo de ejecución**. Los **errores fatales en tiempo de ejecución** hacen que los programas terminen de inmediato, sin haber realizado correctamente su trabajo. Los **errores no fatales en tiempo de ejecución** permiten a los programas ejecutarse hasta terminar su trabajo, lo que a menudo produce resultados incorrectos.*

1.14 Generalidades acerca de Java y este libro

Java es un poderoso lenguaje de programación. En ocasiones, los programadores experimentados se enorgullecen en poder crear un uso excéntrico, deformado e intrincado de un lenguaje. Ésta es una mala práctica de programación; ya que hace que: los programas sean más difíciles de leer, se comporten en forma extraña, sean más difíciles de probar y depurar, y más difíciles de adaptarse a los cambiantes requerimientos. Este libro está enfocado en la *claridad*. A continuación se muestra nuestro primer tip de Buena práctica de programación:



Buena práctica de programación 1.1

Escriba sus programas de Java en una manera simple y directa. A esto se le conoce algunas veces como KIS (Keep It Simple, simplifíquelo). No "extiendas" el lenguaje experimentando con usos excéntricos.

Seguramente habrá escuchado que Java es un lenguaje portable y que los programas escritos en él pueden ejecutarse en diversas computadoras. En general, *la portabilidad es una meta elusiva*.



Tip de portabilidad 1.2

Aunque es más fácil escribir programas portables en Java que en la mayoría de los demás lenguajes de programación, las diferencias entre compiladores, JVMs y computadoras pueden hacer que la portabilidad sea difícil de lograr. No basta con escribir programas en Java para garantizar su portabilidad.



Tip para prevenir errores 1.1

Para asegurarse de que sus programas de Java trabajen correctamente para las audiencias a las que están destinados, pruébelos siempre en todos los sistemas en los que tenga pensado ejecutarlos.

Comparamos nuestra presentación con la documentación de Java de Sun, para verificar que sea completa y precisa. Sin embargo, Java es un lenguaje extenso, y ningún libro puede cubrir todos los temas. En la página `java.sun.com/javase/6/docs/api/index.html` existe una versión de la documentación de las APIs de Java; también puede descargar esta documentación en su propia computadora, visitando `java.sun.com/javase/6/download.jsp`. Para obtener detalles adicionales sobre muchos aspectos del desarrollo en Java, visite `java.sun.com/reference/docs/index.html`.



Buena práctica de programación 1.2

Lea la documentación para la versión de Java que esté utilizando. Consulte esta documentación con frecuencia, para asegurarse de conocer la vasta colección de herramientas disponibles en Java, y para asegurarse de que las está utilizando correctamente.



Buena práctica de programación 1.3

Su computadora y su compilador son buenos maestros. Si, después de leer cuidadosamente el manual de documentación de Java, todavía no está seguro de cómo funciona alguna de sus características, experimente y vea lo que ocurre. Analice cada error o mensaje de advertencia que obtenga al compilar sus programas (a éstos se les llama ***errores en tiempo de compilación*** *o* ***errores de compilación****), y corrija los programas para eliminar estos mensajes.*



Observación de ingeniería de software 1.4

Algunos programadores gustan de leer el código fuente para las clases de la API de Java, para determinar cómo funcionan las clases y aprender técnicas de programación adicionales.

1.15 Prueba de una aplicación en Java

En esta sección, ejecutará su primera aplicación en Java e interactuará con ella. Para empezar, ejecutará una aplicación de ATM, la cual simula las transacciones que se llevan a cabo al utilizar una máquina de cajero automático, o ATM (por ejemplo, retirar dinero, realizar depósitos y verificar los saldos de las cuentas). Aprenderá a crear esta aplicación en el ejemplo práctico opcional orientado a objetos que se incluye en los capítulos 1-8 y 10. La figura 1.10 al final de esta sección sugiere otras aplicaciones interesantes que también puede probar después de terminar con la prueba del ATM. Para los fines de esta sección supondremos que está utilizando Microsoft Windows.

En los siguientes pasos, ejecutará la aplicación y realizará diversas transacciones. Los elementos y la funcionalidad que podemos ver en esta aplicación son típicos de lo que aprenderá a programar en este libro. [*Nota:* utilizamos diversos tipos de letra para diferenciar las características que se ven en una pantalla (por ejemplo, el **Símbolo del sistema**) y los elementos que no se relacionan directamente con una pantalla. Nuestra convención es enfatizar las características de la pantalla como los títulos y menús (por ejemplo, el menú **Archivo**) en una fuente **Helvetica sans-serif** en negritas, y enfatizar los elementos que no son de la pantalla, como los nombres de archivo o los datos de entrada (como `NombrePrograma.java`) en una fuente `Lucida sans-serif`. Como tal vez ya se haya dado cuenta, cuando se ofrece la definición de algún término ésta aparece en negritas. En las figuras en esta sección, resaltamos en gris la entrada del usuario requerida por cada paso, y señalamos las partes importantes de la aplicación con líneas y texto. Para aumentar la visibilidad de estas características, modificamos el color de fondo de las ventanas del **Símbolo del sistema**].

1. ***Revise su configuración.*** Lea la sección *Antes de empezar* para verificar si instaló correctamente Java, y observe si copió los ejemplos del libro en su disco duro.

2 ***Localice la aplicación completa.*** Abra una ventana **Símbolo del sistema**. Para ello, puede seleccionar **Inicio | Todos los programas | Accesorios | Símbolo del sistema.** Cambie al directorio de la aplicación del ATM escribiendo `cd C:\ejemplos\ATM`, y después oprima *Intro* (figura 1.2). El comando `cd` se utiliza para cambiar de directorio.

3. ***Ejecute la aplicación del ATM.*** Escriba el comando `java EjemploPracticoATM` (figura 1.3) oprima *Intro*. Recuerde que el comando `java`, seguido del nombre del archivo `.class` (en este caso, `Ejemplo-PracticoATM`), ejecuta la aplicación. Si especificamos la extensión `.class` al usar el comando `java` se produce un error. [*Nota:* los comandos en Java son sensibles a mayúsculas/minúsculas. Es importante escribir el nombre de esta aplicación con las letras A, T y M mayúsculas en "ATM", una letra E mayúscula en "Ejemplo" y una letra P mayúscula en "Practico". De no ser así, la aplicación no se ejecutará]. Si recibe el mensaje de error `"Exception in thread "main" java.lang.NoClassDefFoundError: EjemploPracticoATM"`, entonces su sistema tiene un problema con `CLASSPATH`. Consulte la sección *Antes de empezar* para obtener instrucciones acerca de cómo corregir este problema.

4. ***Escriba un número de cuenta.*** Cuando la aplicación se ejecuta por primera vez, muestra el mensaje `"Bienvenido!"` y le pide un número de cuenta. Escriba `12345` en el indicador `"Escriba su numero de cuenta:"` (figura 1.4) y oprima *Intro*.

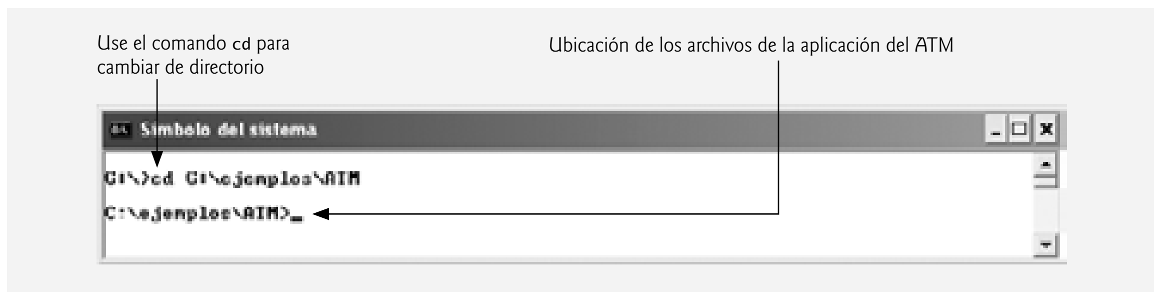


Figura 1.2 | Abrir una ventana **Símbolo del sistema** en Windows XP y cambiar de directorio.



Figura 1.3 | Uso del comando `java` para ejecutar la aplicación del ATM.

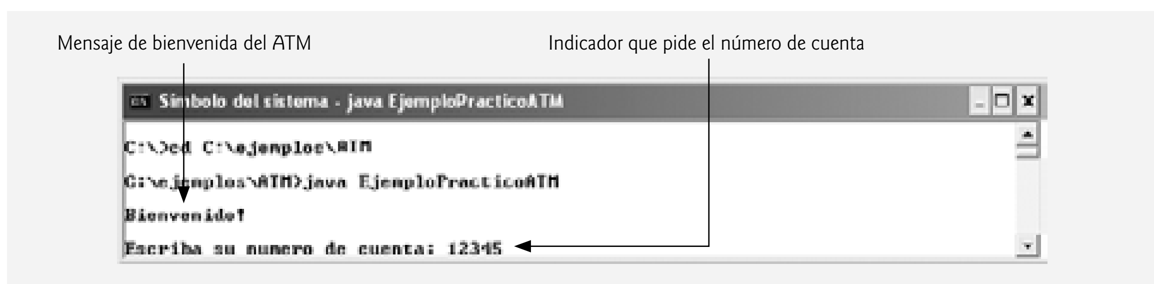


Figura 1.4 | La aplicación pide al usuario un número de cuenta.

5. ***Escriba un NIP.*** Una vez que introduzca un número de cuenta válido, la aplicación mostrará el indicador `"Escriba su NIP:"`. Escriba `"54321"` como su NIP (Número de Identificación Personal) válido y oprima *Intro*. A continuación aparecerá el menú principal del ATM, que contiene una lista de opciones (figura 1.5).

6. ***Revise el saldo de la cuenta.*** Seleccione la opción 1, `"Ver mi saldo"` del menú del ATM (figura 1.6). A continuación la aplicación mostrará dos números: el `Saldo disponible ($1,000.00)` y el `Saldo total ($1,200.00)`. El saldo disponible es la cantidad máxima de dinero en su cuenta, disponible para retirarla en un momento dado. En algunos casos, ciertos fondos como los depósitos recientes, no están disponibles de inmediato para que el usuario pueda retirarlos, por lo que el saldo disponible puede ser menor que el saldo total, como en este caso. Después de mostrar la información de los saldos de la cuenta, se muestra nuevamente el menú principal de la aplicación.

7. ***Retire dinero de la cuenta.*** Seleccione la opción 2, `"Retirar efectivo"`, del menú de la aplicación. A continuación aparecerá (figura 1.7) una lista de montos en dólares (por ejemplo: `20, 40, 60, 100` y `200`). También tendrá la oportunidad de cancelar la transacción y regresar al menú principal. Retire $100 seleccionando la opción `4`. La aplicación mostrará el mensaje `"Tome su efectivo ahora"` y regresará al menú principal. [*Nota:* por desgracia, esta aplicación sólo *simula* el comportamiento de un verdadero ATM, por lo cual no dispensa efectivo en realidad].

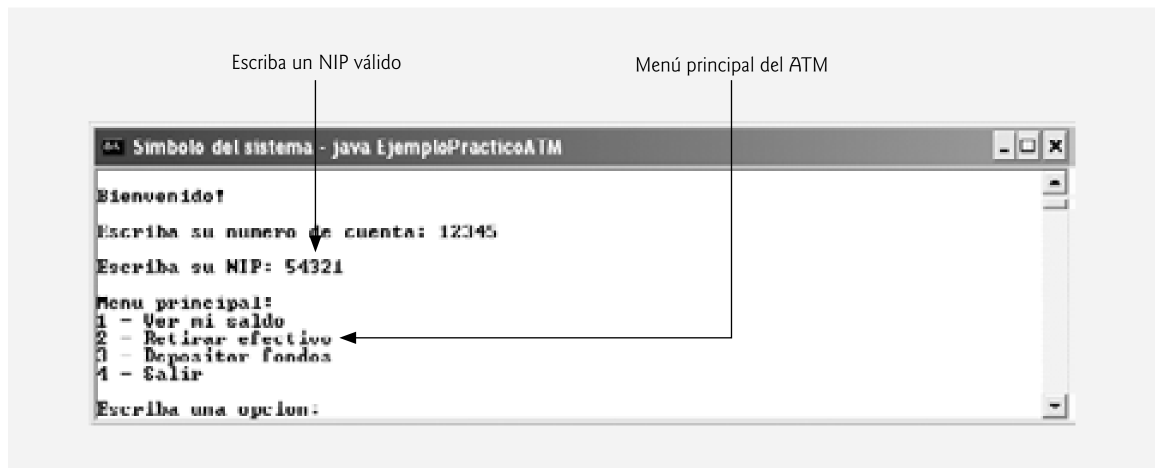


Figura 1.5 | El usuario escribe un número NIP válido y aparece el menú principal de la aplicación del ATM.

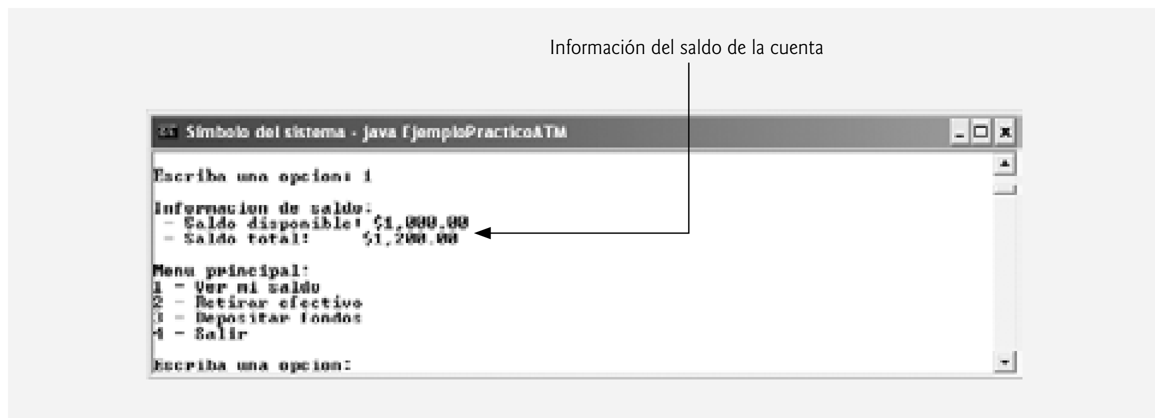


Figura 1.6 | La aplicación del ATM muestra la información del saldo de la cuenta del usuario.

8. ***Confirme que la información de la cuenta se haya actualizado.*** En el menú principal, seleccione la opción `1` nuevamente para ver el saldo actual de su cuenta (figura 1.8). Observe que tanto el saldo disponible como el saldo total se han actualizado para reflejar su transacción de retiro.

9. ***Finalice la transacción.*** Para finalizar su sesión actual en el ATM, seleccione, del menú principal, la opción `4`, `"Salir"` (figura 1.9). El ATM saldrá del sistema y mostrará un mensaje de despedida al usuario. A continuación, la aplicación regresará a su indicador original, pidiendo el número de cuenta del siguiente usuario.

10. ***Salga de la aplicación del ATM y cierre la ventana* Símbolo del sistema.** La mayoría de las aplicaciones cuentan con una opción para salir y regresar al directorio del **Símbolo del sistema** desde el cual se ejecutó la aplicación. Un ATM real no proporciona al usuario la opción de apagar la máquina ATM. En vez de ello, cuando el usuario ha completado todas las transacciones deseadas y elige la opción del menú para salir, el ATM se reinicia a sí mismo y muestra un indicador para el número de cuenta del siguiente

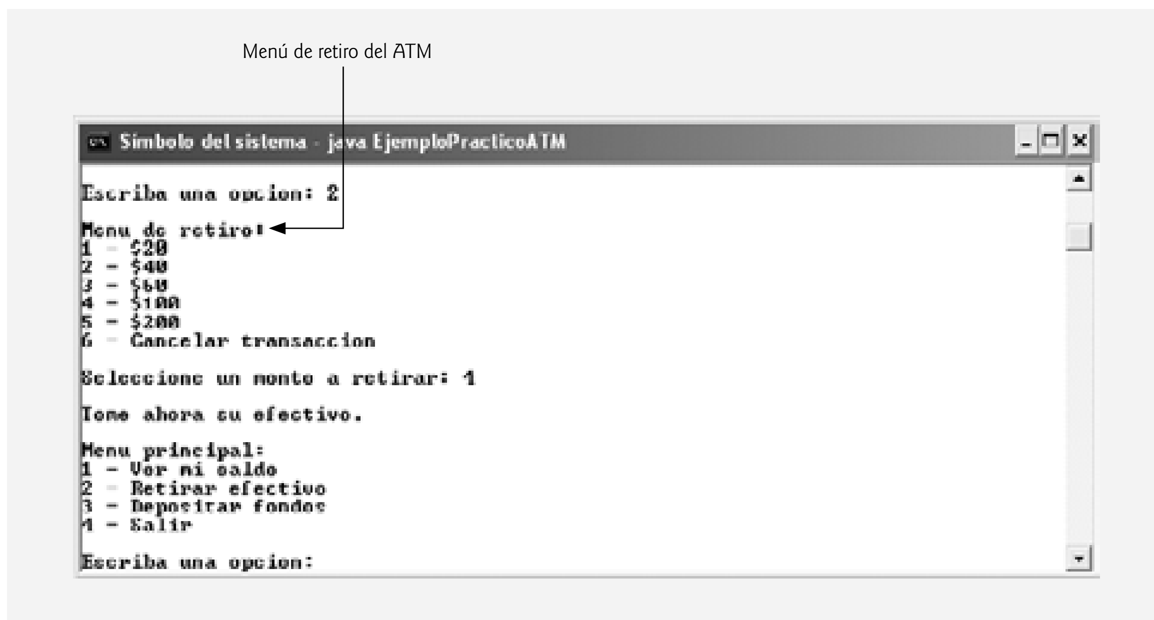


Figura 1.7 | Se retira el dinero de la cuenta y la aplicación regresa al menú principal.

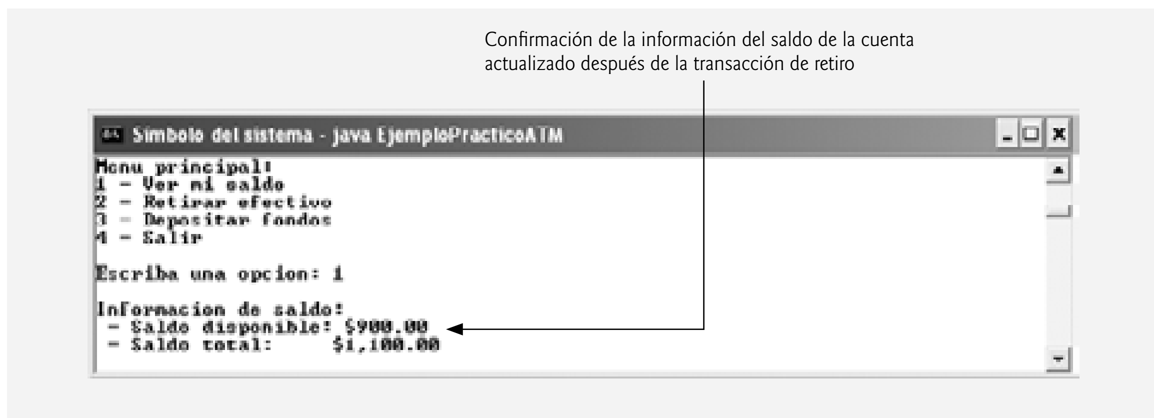


Figura 1.8 | Verificación del nuevo saldo.

usuario. Como se muestra en la figura 1.9, la aplicación del ATM se comporta de manera similar. Al elegir la opción del menú para salir sólo se termina la sesión del usuario actual con el ATM, no toda la aplicación completa. Para salir realmente de la aplicación del ATM, haga clic en el botón Cerrar (**x**) en la esquina superior derecha de la ventana **Símbolo del sistema**. Al cerrar la ventana, la aplicación termina su ejecución.

Aplicaciones adicionales incluidas en Cómo programar en Java, 7ª edición

La figura 1.10 lista unas cuantas de los cientos de aplicaciones que se incluyen en los ejemplos y ejercicios del libro. Estos programas presentan muchas de las poderosas y divertidas características de Java. Ejecute estos programas para que conozca más acerca de las aplicaciones que aprenderá a construir en este libro de texto. La carpeta de ejemplos para este capítulo contiene todos los archivos requeridos para ejecutar cada aplicación. Sólo escriba los comandos que se listan en la figura 1.10 en una ventana de **Símbolo del sistema**.

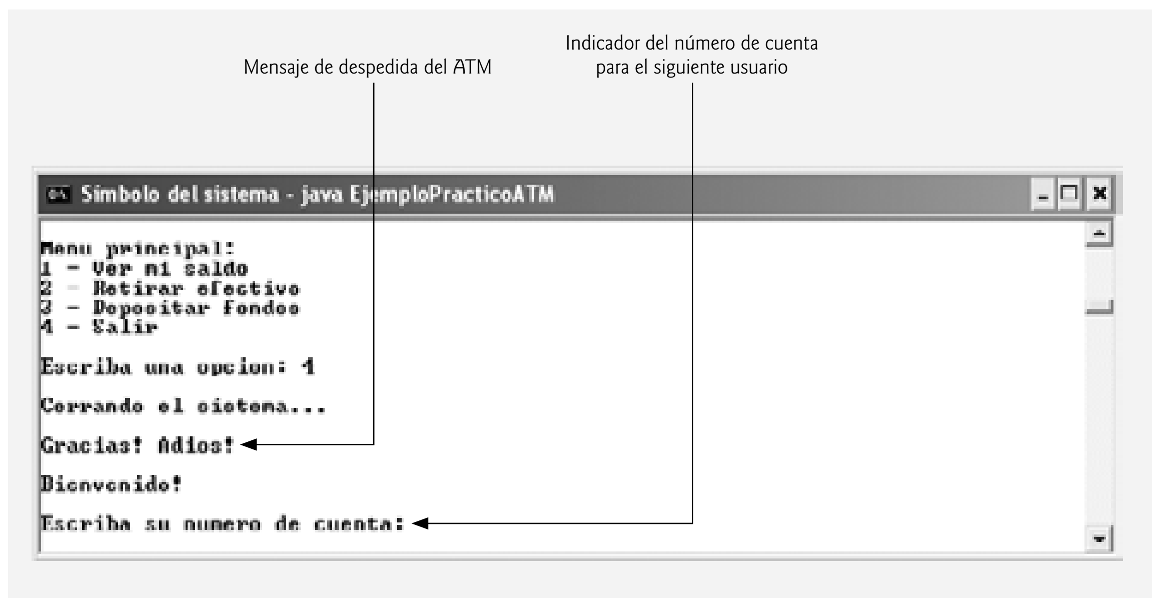


Figura 1.9 | Finalización de una sesión de transacciones con el ATM.

Nombre de la aplicación	Capítulo(s) en donde se ubica	Comandos a ejecutar
Tic-Tac-Toe	Capítulos 8 y 24	`cd C:\ejemplos\cap01\Tic-Tac-Toe` `Java PruebaTicTacToe`
Juego de adivinanza	Capítulo 11	`cd C:\ejemplos\cap01\JuegoAdivinanza` `Java JuegoAdivinanza`
Animador de logotipos	Capítulo 21	`cd C:\ejemplos\cap01\AnimadorLogotipos` `Java AnimadorLogotipos`
Pelota rebotadora	Capítulo 23	`cd C:\ejemplos\cap01\PelotaRebotadora` `Java PelotaRebotadora`

Figura 1.10 | Ejemplos de aplicaciones de Java adicionales, incluidas en *Cómo programar en Java, 7ª edición.*

1.16 Ejemplo práctico de Ingeniería de Software: introducción a la tecnología de objetos y UML

Ahora empezaremos nuestra primera introducción al tema de la orientación a objetos, una manera natural de pensar acerca del mundo real y de escribir programas de cómputo. Los capítulos 1-8 y 10 terminan con una sección breve titulada Ejemplo práctico de Ingeniería de Software, en la cual presentamos una introducción cuidadosamente guiada al tema de la orientación a objetos. Nuestro objetivo aquí es ayudarle a desarrollar una forma de pensar orientada a objetos, y de presentarle el **Lenguaje Unificado de Modelado™ (UML™)**, un lenguaje gráfico que permite a las personas que diseñan sistemas de software utilizar una notación estándar en la industria para representarlos.

En esta única sección requerida (1.16), presentamos los conceptos y la terminología de la orientación a objetos. Las secciones opcionales en los capítulos 2-8 y 10 presentan el diseño y la implementación orientados a objetos de un software para una máquina de cajero automático (ATM) simple. Las secciones tituladas Ejemplo práctico de Ingeniería de Software al final de los capítulos 2 al 7:

- analizan un documento de requerimientos típico que describe un sistema de software (el ATM) que construirá
- determinan los objetos requeridos para implementar ese sistema
- establecen los atributos que deben tener estos objetos
- fijan los comportamientos que exhibirán estos objetos
- especifican la forma en que los objetos deben interactuar entre sí para cumplir con los requerimientos del sistema

Las secciones tituladas Ejemplo práctico de Ingeniería de Software al final de los capítulos 8 y 10 modifican y mejoran el diseño presentado en los capítulos 2 al 7. El apéndice M contiene una implementación completa y funcional en Java del sistema ATM orientado a objetos.

Usted experimentará una concisa pero sólida introducción al diseño orientado a objetos con UML. Además, afinará sus habilidades para leer código al ver paso a paso la implementación del ATM en Java, cuidadosamente escrita y bien documentada, en el apéndice M.

Conceptos básicos de la tecnología de objetos

Comenzaremos nuestra introducción al tema de la orientación a objetos con cierta terminología clave. En cualquier parte del mundo real puede ver **objetos**: gente, animales, plantas, automóviles, aviones, edificios, computadoras, etcétera. Los humanos pensamos en términos de objetos. Los teléfonos, casas, semáforos, hornos de microondas y enfriadores de agua son sólo unos cuantos objetos más. Los programas de cómputo, como los programas de Java que leerá en este libro y los que usted mismo escriba, están compuestos por muchos objetos de software con capacidad de interacción.

En ocasiones dividimos a los objetos en dos categorías: animados e inanimados. Los objetos animados están "vivos" en cierto sentido; se mueven a su alrededor y hacen cosas. Por otro lado, los objetos inanimados no se mueven por su propia cuenta. Sin embargo, los objetos de ambos tipos tienen ciertas cosas en común. Todos ellos tienen **atributos** (como tamaño, forma, color y peso), y todos exhiben **comportamientos** (por ejemplo, una pelota rueda, rebota, se infla y desinfla; un bebé llora, duerme, gatea, camina y parpadea; un automóvil acelera, frena y da vuelta; una toalla absorbe agua). Estudiaremos los tipos de atributos y comportamientos que tienen los objetos de software.

Los humanos aprenden acerca de los objetos existentes estudiando sus atributos y observando sus comportamientos. Distintos objetos pueden tener atributos similares y pueden exhibir comportamientos similares. Por ejemplo, pueden hacerse comparaciones entre los bebés y los adultos, y entre los humanos y los chimpancés.

El **diseño orientado a objetos (DOO)** modela el software en términos similares a los que utilizan las personas para describir objetos del mundo real. Este diseño aprovecha las relaciones entre las **clases**, en donde los objetos de cierta clase (como una clase de vehículos) tienen las mismas características; los automóviles, camiones, pequeños vagones rojos y patines tienen mucho en común. El DOO también aprovecha las relaciones de **herencia**, en donde las nuevas clases de objetos se derivan absorbiendo las características de las clases existentes y agregando sus propias características únicas. Un objeto de la clase "convertible" ciertamente tiene las características de la clase más general "automóvil" pero, de manera más específica, el techo de un convertible puede ponerse y quitarse.

El diseño orientado a objetos proporciona una manera natural e intuitiva de ver el proceso de diseño de software: a saber, modelando los objetos por sus atributos y comportamientos, de igual forma que como describimos los objetos del mundo real. El DOO también modela la comunicación entre los objetos. Así como las personas se envían mensajes unas a otras (por ejemplo, un sargento ordenando a un soldado que permanezca firme), los objetos también se comunican mediante mensajes. Un objeto cuenta de banco puede recibir un mensaje para reducir su saldo por cierta cantidad, debido a que el cliente ha retirado esa cantidad de dinero.

El DOO **encapsula** (es decir, envuelve) los atributos y las **operaciones** (comportamientos) en los objetos; los atributos y las operaciones de un objeto se enlazan íntimamente entre sí. Los objetos tienen la propiedad de **ocultamiento de información**. Esto significa que los objetos pueden saber cómo comunicarse entre sí a través de **interfaces** bien definidas, pero por lo general no se les permite saber cómo se implementan otros objetos; los detalles de la implementación se ocultan dentro de los mismos objetos. Por ejemplo, podemos conducir un automóvil con efectividad, sin necesidad de saber los detalles acerca de cómo funcionan internamente los motores, las transmisiones y los sistemas de escape; siempre y cuando sepamos cómo usar el pedal del acelerador, el pedal del freno, el volante, etcétera. Más adelante veremos por qué el ocultamiento de información es tan imprescindible para la buena ingeniería de software.

Los lenguajes como Java son **orientados a objetos**. La programación en dichos lenguajes se llama **programación orientada a objetos (POO)**, y permite a los programadores de computadoras implementar un diseño orientado a objetos como un sistema funcional. Por otra parte, los lenguajes como C son **por procedimientos**, de manera que la programación tiende a ser **orientada a la acción**. En C, la unidad de programación es la **función**. Los grupos de acciones que realizan cierta tarea común se forman en funciones, y las funciones se agrupan para formar programas. En Java, la unidad de programación es la clase a partir de la cual se **instancian** (crean) los objetos en un momento dado. Las clases en Java contienen **métodos** (que implementan operaciones y son similares a las funciones en C) y **campos** (que implementan atributos).

Los programadores de Java se concentran en crear clases. Cada clase contiene campos, además del conjunto de métodos que manipulan esos campos y proporcionan servicios a **clientes** (es decir, otras clases que utilizan esa clase). El programador utiliza las clases existentes como bloques de construcción para crear nuevas clases.

Las clases son para los objetos lo que los planos de construcción, para las casas. Así como podemos construir muchas casas a partir de un plano, podemos instanciar (crear) muchos objetos a partir de una clase. No puede cocinar alimentos en la cocina de un plano de construcción; puede cocinarlos en la cocina de una casa.

Las clases pueden tener relaciones con otras clases. Por ejemplo, en un diseño orientado a objetos de un banco, la clase "cajero" necesita relacionarse con las clases "cliente", "cajón de efectivo", "bóveda", etcétera. A estas relaciones se les llama **asociaciones**.

Al empaquetar el software en forma de clases, los sistemas de software posteriores pueden **reutilizar** esas clases. Los grupos de clases relacionadas se empaquetan comúnmente como **componentes** reutilizables. Así como los corredores de bienes raíces dicen a menudo que los tres factores más importantes que afectan el precio de los bienes raíces son "la ubicación, la ubicación y la ubicación", las personas en la comunidad de software dicen a menudo que los tres factores más importantes que afectan el futuro del desarrollo de software son "la reutilización, la reutilización y la reutilización". Reutilizar las clases existentes cuando se crean nuevas clases y programas es un proceso que ahorra tiempo y esfuerzo; también ayuda a los programadores a crear sistemas más confiables y efectivos, ya que las clases y componentes existentes a menudo han pasado por un proceso extenso de prueba, depuración y optimización del rendimiento.

Evidentemente, con la tecnología de objetos podemos crear la mayoría del software que necesitaremos mediante la combinación de clases, así como los fabricantes de automóviles combinan las piezas intercambiables. Cada nueva clase que usted cree tendrá el potencial de convertirse en una valiosa pieza de software, que usted y otros programadores podrán usar para agilizar y mejorar la calidad de los futuros esfuerzos de desarrollo de software.

Introducción al análisis y diseño orientados a objetos (A/DOO)

Pronto estará escribiendo programas en Java. ¿Cómo creará el código para sus programas? Tal vez, como muchos programadores principiantes, simplemente encenderá su computadora y empezará a teclear. Esta metodología puede funcionar para programas pequeños (como los que presentamos en los primeros capítulos del libro) pero ¿qué haría usted si se le pidiera crear un sistema de software para controlar miles de máquinas de cajero automático para un importante banco? O suponga que le pidieron trabajar en un equipo de 1,000 desarrolladores de software para construir el nuevo sistema de control de tráfico aéreo de Estados Unidos. Para proyectos tan grandes y complejos, no podría simplemente sentarse y empezar a escribir programas.

Para crear las mejores soluciones, debe seguir un proceso detallado para **analizar** los **requerimientos** de su proyecto (es decir, determinar *qué* es lo que se supone debe hacer el sistema) y desarrollar un **diseño** que cumpla con esos requerimientos (es decir, decidir *cómo* debe hacerlo el sistema). Idealmente usted pasaría por este proceso y revisaría cuidadosamente el diseño (o haría que otros profesionales de software lo revisaran) antes de escribir cualquier código. Si este proceso implica analizar y diseñar su sistema desde un punto de vista orientado a objetos, lo llamamos un **proceso de análisis y diseño orientado a objetos (A/DOO)**. Los programadores experimentados saben que el análisis y el diseño pueden ahorrar innumerables horas, ya que les ayudan a evitar un método de desarrollo de un sistema mal planeado, que tiene que abandonarse en plena implementación, con la posibilidad de desperdiciar una cantidad considerable de tiempo, dinero y esfuerzo.

A/DOO es el término genérico para el proceso de analizar un problema y desarrollar un método para resolverlo. Los pequeños problemas como los que se describen en los primeros capítulos de este libro no requieren de un proceso exhaustivo de A/DOO. Podría ser suficiente con escribir **pseudocódigo** antes de empezar a escribir el código en Java; el pseudocódigo es un medio informal para expresar la lógica de un programa. En realidad no es un lenguaje de programación, pero podemos usarlo como un tipo de bosquejo para guiarnos mientras escribimos nuestro código. En el capítulo 4 presentamos el pseudocódigo.

A medida que los problemas y los grupos de personas que los resuelven aumentan en tamaño, los métodos de A/DOO se vuelven más apropiados que el pseudocódigo. Idealmente, un grupo debería acordar un proceso estrictamente definido para resolver su problema, y establecer también una manera uniforme para que los miembros del grupo se comuniquen los resultados de ese proceso entre sí. Aunque existen diversos procesos de A/DOO, hay un lenguaje gráfico para comunicar los resultados de *cualquier* proceso A/DOO que se ha vuelto muy popular. Este lenguaje, conocido como Lenguaje Unificado de Modelado (UML), se desarrolló a mediados de la década de los noventa, bajo la dirección inicial de tres metodologistas de software: Grady Booch, James Rumbaugh e Ivar Jacobson.

Historia de UML

En la década de los ochenta, un creciente número de empresas comenzó a utilizar la POO para crear sus aplicaciones, lo cual generó la necesidad de un proceso estándar de A/DOO. Muchos metodologistas (incluyendo a Booch, Rumbaugh y Jacobson) produjeron y promocionaron, por su cuenta, procesos separados para satisfacer esta necesidad. Cada uno de estos procesos tenía su propia notación, o “lenguaje” (en forma de diagramas gráficos), para transmitir los resultados del análisis y el diseño.

A principios de la década de los noventa, diversas compañías (e inclusive diferentes divisiones dentro de la misma compañía) utilizaban sus propios procesos y notaciones únicos. Al mismo tiempo, estas compañías querían utilizar herramientas de software que tuvieran soporte para sus procesos particulares. Con tantos procesos, se les dificultó a los distribuidores de software proporcionar dichas herramientas. Evidentemente era necesario contar con una notación y procesos estándar.

En 1994, James Rumbaugh se unió con Grady Booch en Rational Software Corporation (ahora una división de IBM), y comenzaron a trabajar para unificar sus populares procesos. Pronto se unió a ellos Ivar Jacobson. En 1996, el grupo liberó las primeras versiones de UML para la comunidad de ingeniería de software, solicitando retroalimentación. Casi al mismo tiempo, una organización conocida como **Object Management Group™ (OMG™, Grupo de administración de objetos)** hizo una invitación para participar en la creación de un lenguaje común de modelado. El OMG (`www.omg.org`) es una organización sin fines de lucro que promueve la estandarización de las tecnologías orientadas a objetos, emitiendo lineamientos y especificaciones como UML. Varias empresas (entre ellas HP, IBM, Microsoft, Oracle y Rational Software) habían reconocido ya la necesidad de un lenguaje común de modelado. Estas compañías formaron el consorcio **UML Partners (Socios de UML)** en respuesta a la solicitud de proposiciones por parte del OMG (el consorcio que desarrolló la versión 1.1 de UML y la envió al OMG). La propuesta fue aceptada y, en 1997, el OMG asumió la responsabilidad del mantenimiento y revisión de UML en forma continua. La versión 2 que está ahora disponible marca la primera modificación importante al UML desde el estándar de la versión 1.1 de 1997. A lo largo de este libro, presentaremos la terminología y notación de UML 2.

¿Qué es UML?

UML es ahora el esquema de representación gráfica más utilizado para modelar sistemas orientados a objetos. Evidentemente ha unificado los diversos esquemas de notación populares. Aquellos quienes diseñan sistemas utilizan el lenguaje (en forma de diagramas) para modelar sus sistemas.

Una característica atractiva es su flexibilidad. UML es **extensible** (es decir, capaz de mejorarse con nuevas características) e independiente de cualquier proceso de A/DOO específico. Los modeladores de UML tienen la libertad de diseñar sistemas utilizando varios procesos, pero todos los desarrolladores pueden ahora expresar esos diseños con un conjunto de notaciones gráficas estándar.

UML es un lenguaje gráfico complejo, con muchas características. En nuestras secciones del Ejemplo práctico de Ingeniería de Software, presentamos un subconjunto conciso y simplificado de estas características. Luego utilizamos este subconjunto para guiarlo a través de la experiencia de su primer diseño con UML, la cual está dirigida a los programadores principiantes orientados a objetos en cursos de programación de primer o segundo semestre.

Recursos Web de UML

Para obtener más información acerca de UML, consulte los siguientes sitios Web:

`www.uml.org`

Esta página de recursos de UML del Grupo de Administración de Objetos (OMG) proporciona documentos de la especificación para UML y otras tecnologías orientadas a objetos.

`www.ibm.com/software/rational/uml`

Ésta es la página de recursos de UML para IBM Rational, sucesor de Rational Software Corporation (la compañía que creó a UML).

`en.wikipedia.org/wiki/UML`

La definición de Wikipedia de UML. Este sitio también ofrece vínculos a muchos recursos adicionales de UML.

`es.wikipedia.org/wiki/UML`

La definición de Wikipedia del UML en español.

Lecturas recomendadas

Los siguientes libros proporcionan información acerca del diseño orientado a objetos con UML:

Ambler, S. *The Object Primer: Agile Model-Driven Development with UML 2.0, Third Edition*. Nueva York: Cambridge University Press, 2005.

Arlow, J. e I. Neustadt. *UML and the Unified Process: Practical Object-Oriented Analysis and Design, Second Edition*. Boston: Addison-Wesley Professional, 2006.

Fowler, M. *UML Distilled, Third Edition: A Brief Guide to the Standard Object Modeling Language*. Boston: Addison-Wesley Professional, 2004.

Rumbaugh, J., I. Jacobson y G. Booch. *The Unified Modeling Language User Guide, Second Edition*. Boston: Addison-Wesley Professional, 2006.

Ejercicios de autorrepaso de la sección 1.16

1.1 Liste tres ejemplos de objetos reales que no mencionamos. Para cada objeto, liste varios atributos y comportamientos.

1.2 El pesudocódigo es __________.
- a) otro término para el A/DOO
- b) un lenguaje de programación utilizado para visualizar diagramas de UML
- c) un medio informal para expresar la lógica de un programa
- d) un esquema de representación gráfica para modelar sistemas orientados a objetos

1.3 El UML se utiliza principalmente para __________.
- a) probar sistemas orientados a objetos
- b) diseñar sistemas orientados a objetos
- c) implementar sistemas orientados a objetos
- d) a y b

Respuestas a los ejercicios de autorrepaso de la sección 1.16

1.1 [*Nota:* las respuestas pueden variar]. a) Los atributos de una televisión incluyen el tamaño de la pantalla, el número de colores que puede mostrar, su canal actual y su volumen actual. Una televisión se enciende y se apaga, cam-

bia de canales, muestra video y reproduce sonidos. b) Los atributos de una cafetera incluyen el volumen máximo de agua que puede contener, el tiempo requerido para preparar una jarra de café y la temperatura del plato calentador bajo la jarra de café. Una cafetera se enciende y se apaga, prepara café y lo calienta. c) Los atributos de una tortuga incluyen su edad, el tamaño de su caparazón y su peso. Una tortuga camina, se mete en su caparazón, emerge del mismo y come vegetación.

1.2 c.

1.3 b.

1.17 Web 2.0

Literalmente, la Web explotó a mediados de la década de los noventa, pero surgieron tiempos difíciles a principios del año 2000, debido al desplome económico de punto com. Al resurgimiento que empezó aproximadamente en el 2004, se le conoce como Web 2.0. La primera Conferencia sobre Web 2.0 se realizó en el 2004. Un año después, el término "Web 2.0" obtuvo aproximadamente 10 millones de coincidencias en el motor de búsqueda **Google**, para crecer hasta 60 millones al año siguiente. A Google se le considera en muchas partes como la compañía característica de Web 2.0. Algunas otras son Craigslist (listados gratuitos de anuncios clasificados), Flickr (sitio para compartir fotos), del.icio.us (sitios favoritos de carácter social), YouTube (sitio para compartir videos), MySpace y FaceBook (redes sociales), Salesforce (software de negocios que se ofrece como servicio en línea), Second Life (un mundo virtual), Skype (telefonía por Internet) y Wikipedia (una enciclopedia en línea gratuita).

En Deitel & Associates, inauguramos nuestra **Iniciativa de Negocios por Internet** basada en Web 2.0 en el año 2005. Estamos investigando las tecnologías clave de Web 2.0 y las utilizamos para crear negocios en Internet. Compartimos nuestra investigación en forma de Centros de recursos en `www.deitel.com/resourcecenters.html`. Cada semana anunciamos los Centros de recursos más recientes en nuestro boletín de correo electrónico *Deitel® Buzz Online* (`www.deitel.com/newsletter/subscribe.html`). Cada uno de estos centros lista muchos vínculos a contenido y software gratuito en Internet.

En este libro incluimos un tratamiento detallado sobre los **servicios Web** (capítulo 28) y presentamos la nueva metodología de desarrollo de aplicaciones, conocida como **mashups** (apéndice H), en la que puede desarrollar rápidamente aplicaciones poderosas e intrigantes, al combinar servicios Web complementarios y otras fuentes de información provenientes de dos o más organizaciones. Un mashup popular es `www.housingmaps.com`, el cual combina los listados de bienes raíces de `www.craigslist.org` con las capacidades de los mapas de **Google Maps** para mostrar las ubicaciones de los apartamentos para renta en un área dada.

Ajax es una de las tecnologías más importantes de Web 2.0. Aunque el uso del término explotó en el 2005, es sólo un término que nombra a un grupo de tecnologías y técnicas de programación que han estado en uso desde finales de la década de los noventa. Ajax ayuda a las aplicaciones basadas en Internet a funcionar como las aplicaciones de escritorio; una tarea difícil, dado que dichas aplicaciones sufren de retrasos en la transmisión, a medida que los datos se intercambian entre su computadora y las demás computadoras en Internet. Mediante el uso de Ajax, las aplicaciones como Google Maps han logrado un desempeño excelente, además de la apariencia visual de las aplicaciones de escritorio. Aunque no hablaremos sobre la programación "pura" con Ajax en este libro (que es bastante compleja), en el capítulo 27 mostraremos cómo crear aplicaciones habilitadas para Ajax mediante el uso de los componentes de JavaServer Faces (JSF) habilitados para Ajax.

Los **blogs** son sitios Web (actualmente hay como 60 millones de ellos) similares a un diario en línea, en donde las entradas más recientes aparecen primero. Los "bloggers" publican rápidamente sus opiniones acerca de las noticias, lanzamientos de productos, candidatos políticos, temas controversiales, y de casi todo lo demás. A la colección de todos los blogs y de la comunidad de "blogging" se le conoce como **blogósfera**, y cada vez está teniendo más influencia. **Technorati** es el líder en motores de búsqueda de blogs.

Las **fuentes RSS** permiten a los sitios enviar información a sus suscriptores. Un uso común de las fuentes RSS es enviar las publicaciones más recientes de los blogs, a las personas que se suscriben a éstos. Los flujos de información RSS en Internet están creciendo de manera exponencial.

Web 3.0 es otro nombre para la siguiente generación de la Web, que también se le conoce como **Web Semántica**. Casi todo el contenido de Web 1.0 estaba basado en HTML. Web 2.0 está utilizando cada vez más el XML, en especial en tecnologías como las fuentes RSS. Web 3.0 utilizará aún más el XML, creando una "Web de significado". Si usted es un estudiante que busca un excelente artículo de presentación o un tema para una tesis, o si es un emprendedor que busca oportunidades de negocios, dé un vistazo a nuestro Centro de recursos sobre Web 3.0.

Para seguir los últimos desarrollos en Web 2.0, visite www.techcrunch.com y www.slashdot.org, y revise la lista creciente de Centros de recursos relacionados con Web 2.0 en www.deitel.com/resourcecenters.html.

1.18 Tecnologías de software

En esta sección hablaremos sobre varias "palabras de moda" que escuchará en la comunidad de desarrollo de software. Creamos Centros de recursos sobre la mayoría de estos temas, y hay muchos por venir.

Agile Software Development (Desarrollo Ágil de Software) es un conjunto de metodologías que tratan de implementar software rápidamente, con menos recursos que las metodologías anteriores. Visite los sitios de Agile Alliance (www.agilealliance.org) y Agile Manifesto (www.agilemanifesto.org). También puede visitar el sitio en español www.agile-spain.com.

Extreme programming (XP) (Programación extrema (PX)) es una de las diversas tecnologías de desarrollo ágil. Trata de desarrollar software con rapidez. El software se libera con frecuencia en pequeños incrementos, para alentar la rápida retroalimentación de los usuarios. PX reconoce que los requerimientos de los usuarios cambian a menudo, y que el software debe cumplir con esos requerimientos rápidamente. Los programadores trabajan en pares en una máquina, de manera que la revisión del código se realiza de inmediato, a medida que se crea el código. Todos en el equipo deben poder trabajar con cualquier parte del código.

Refactoring (Refabricación) implica la reformulación del código para hacerlo más claro y fácil de mantener, al tiempo que se preserva su funcionalidad. Se emplea ampliamente con las metodologías de desarrollo ágil. Hay muchas herramientas de refabricación disponibles para realizar las porciones principales de la reformulación de manera automática.

Los **patrones de diseño** son arquitecturas probadas para construir software orientado a objetos flexible y que pueda mantenerse (vea el apéndice P Web adicional). El campo de los patrones de diseño trata de enumerar a los patrones recurrentes, y de alentar a los diseñadores de software para que los reutilicen y puedan desarrollar un software de mejor calidad con menos tiempo, dinero y esfuerzo.

Programación de juegos. El negocio de los juegos de computadora es más grande que el negocio de las películas de estreno. Ahora hay cursos universitarios, e incluso maestrías, dedicados a las técnicas sofisticadas de software que se utilizan en la programación de juegos. Vea nuestros Centros de recursos sobre Programación de juegos y Proyectos de programación.

El **software de código fuente abierto** es un estilo de desarrollo de software que contrasta con el desarrollo propietario, que dominó los primeros años del software. Con el desarrollo de código fuente abierto, individuos y compañías contribuyen sus esfuerzos en el desarrollo, mantenimiento y evolución del software, a cambio del derecho de usar ese software para sus propios fines, comúnmente sin costo. Por lo general, el código fuente abierto se examina a detalle por una audiencia más grande que el software propietario, por lo cual los errores se eliminan con más rapidez. El código fuente abierto también promueve más innovación. Sun anunció recientemente que piensa abrir el código fuente de Java. Algunas de las organizaciones de las que se habla mucho en la comunidad de código fuente abierto son Eclipse Foundation (el IDE de Eclipse es popular para el desarrollo de software en Java), Mozilla Foundation (creadores del explorador Web Firefox), Apache Software Foundation (creadores del servidor Web Apache) y SourceForge (que proporciona las herramientas para administrar proyectos de código fuente abierto y en la actualidad cuenta con más de 100,000 proyectos en desarrollo).

Linux es un sistema operativo de código fuente abierto, y uno de los más grandes éxitos de la iniciativa de código fuente abierto. **MySQL** es un sistema de administración de bases de datos con código fuente abierto. **PHP** es el lenguaje de secuencias de comandos del lado servidor para Internet de código fuente abierto más popular, para el desarrollo de aplicaciones basadas en Internet. **LAMP** es un acrónimo para el conjunto de tecnologías de código fuente abierto que utilizaron muchos desarrolladores para crear aplicaciones Web: representa a Linux, Apache, MySQL y PHP (o Perl, o Python; otros dos lenguajes que se utilizan para propósitos similares).

Ruby on Rails combina el lenguaje de secuencias de comandos Ruby con el marco de trabajo para aplicaciones Web Rails, desarrollado por la compañía 37Signals. Su libro, *Getting Real*, es una lectura obligatoria para los desarrolladores de aplicaciones Web de la actualidad; puede leerlo sin costo en gettingreal.37signals.com/toc.php. Muchos desarrolladores de Ruby on Rails han reportado un considerable aumento en la productividad, en comparación con otros lenguajes al desarrollar aplicaciones Web con uso intensivo de bases de datos.

Por lo general, el software siempre se ha visto como un producto; la mayoría del software aún se ofrece de esta manera. Si desea ejecutar una aplicación, compra un paquete de software de un distribuidor. Después instala ese software en su computadora y lo ejecuta según sea necesario. Al aparecer nuevas versiones del software, usted lo

actualiza, a menudo con un costo considerable. Este proceso puede volverse incómodo para empresas con decenas de miles de sistemas que deben mantenerse en una extensa colección de equipo de cómputo. Con **Software as a Service (SAAS)**, el software se ejecuta en servidores ubicados en cualquier parte de Internet. Cuando se actualiza ese servidor, todos los clientes a nivel mundial ven las nuevas características; no se necesita instalación local. Usted accede al servidor a través de un explorador Web; éstos son bastante portables, por lo que puede ejecutar las mismas aplicaciones en distintos tipos de computadoras, desde cualquier parte del mundo. Salesforce.com, Google, Microsoft Office Live y Windows Live ofrecen SAAS.

1.19 Conclusión

Este capítulo presentó los conceptos básicos de hardware y software, y los conceptos de la tecnología básica de objetos, incluyendo clases, objetos, atributos, comportamientos, encapsulamiento, herencia y polimorfismo. Hablamos sobre los distintos tipos de lenguajes de programación y cuáles son los más utilizados. Conoció los pasos para crear y ejecutar una aplicación de Java mediante el uso del JDK 6 de Sun. El capítulo exploró la historia de Internet y World Wide Web, y la función de Java en cuanto al desarrollo de aplicaciones cliente/servidor distribuidas para Internet y Web. También aprendió acerca de la historia y el propósito de UML: el lenguaje gráfico estándar en la industria para modelar sistemas de software. Por último, realizó pruebas de una o más aplicaciones de Java, similares a los tipos de aplicaciones que aprenderá a programar en este libro.

En el capítulo 2 creará sus primeras aplicaciones en Java. Verá ejemplos que muestran cómo los programas imprimen mensajes en pantalla y obtienen información del usuario para procesarla. Analizaremos y explicaremos cada ejemplo, para facilitarle el proceso de aprender a programar en Java.

1.20 Recursos Web

Esta sección proporciona muchos recursos que le serán de utilidad a medida que aprenda Java. Los sitios incluyen recursos de Java, herramientas de desarrollo de Java para estudiantes y profesionales, y nuestros propios sitios Web, en donde podrá encontrar descargas y recursos asociados con este libro. También le proporcionaremos un vínculo, en donde podrá suscribirse a nuestro boletín de correo electrónico *Deitel® Buzz Online* sin costo.

Sitios Web de Deitel & Associates

`www.deitel.com`

Contiene actualizaciones, correcciones y recursos adicionales para todas las publicaciones Deitel.

`www.deitel.com/newsletter/subscribe.html`

Suscríbase al boletín de correo electrónico gratuito *Deitel® Buzz Online*, para seguir el programa de publicaciones de Deitel & Associates, incluyendo actualizaciones y fe de erratas para este libro.

`www.prenhall.com/deitel`

La página de inicio de Prentice Hall para las publicaciones Deitel. Aquí encontrará información detallada sobre los productos, capítulos de ejemplo y *Sitios Web complementarios* con recursos para estudiantes e instructores.

`www.deitel.com/books/jhtp7/`

La página de inicio de Deitel & Associates para *Cómo programar en Java, 7ª edición*. Aquí encontrará vínculos a los ejemplos del libro (que también se incluyen en el CD que viene con el libro) y otros recursos.

Centros de recursos de Deitel sobre Java

`www.deitel.com/Java/`

Nuestro Centro de recursos sobre Java se enfoca en la enorme cantidad de contenido gratuito sobre Java, disponible en línea. Empiece aquí su búsqueda de recursos, descargas, tutoriales, documentación, libros, libros electrónicos, diarios, artículos, blogs y más, que le ayudarán a crear aplicaciones en Java.

`www.deitel.com/JavaSE6Mustang/`

Nuestro Centro de recursos sobre Java SE 6 (Mustang) es su guía para la última versión de Java. Este sitio incluye los mejores recursos que encontramos en línea, para ayudarle a empezar con el desarrollo en Java SE 6.

`www.deitel.com/JavaEE5/`

Nuestro Centro de recursos sobre Java Enterprise Edition 5 (Java EE 5).

`www.deitel.com/JavaCertification/`

Nuestro Centro de recursos de evaluación de certificación y valoración.

`www.deitel.com/JavaDesignPatterns/`
Nuestro Centro de recursos sobre los patrones de diseño de Java. En su libro, *Design Patterns: Elements of Reusable Object-Oriented Software* (Boston: Addison-Wesley Professional, 1995), la "Banda de los cuatro" (E. Gamma, R. Helm, R. Jonson y J. Vlissides) describen 23 patrones de diseño que proporcionan arquitecturas demostradas para construir sistemas de software orientados a objetos. En este centro de recursos, encontrará discusiones sobre muchos de éstos y otros patrones de diseño.

`www.deitel.com/CodeSearchEngines/`
Nuestro Centro de recursos sobre Motores de Búsqueda de Código y Sitios de Código incluye recursos que los desarrolladores utilizan para buscar código fuente en línea.

`www.deitel.com/ProgrammingProjects/`
Nuestro Centro de recursos sobre Proyectos de Programación es su guía para proyectos de programación estudiantiles en línea.

Sitios Web de Sun Microsystems

`java.sun.com/developer/onlineTraining/new2java/index.html`
El centro "New to Java Center" (Centro para principiantes en Java) en el sitio Web de Sun Microsystems ofrece recursos de capacitación en línea para ayudarle a empezar con la programación en Java.

`java.sun.com/javase/6/download.jsp`
La página de descarga para el Kit de Desarrollo de Java 6 (JDK 6) y su documentación. El JDK incluye todo lo necesario para compilar y ejecutar sus aplicaciones en Java SE 6 (Mustang).

`java.sun.com/javase/6/webnotes/install/index.html`
Instrucciones para instalar el JDK 6 en plataformas Solaris, Windows y Linux.

`java.sun.com/javase/6/docs/api/index.html`
El sitio en línea para la documentación de la API de Java SE 6.

`java.sun.com/javase`
La página de inicio para la plataforma Java Standard Edition.

`java.sun.com`
La página de inicio de la tecnología Java de Sun ofrece descargas, referencias, foros, tutoriales en línea y mucho más.

`java.sun.com/reference/docs/index.html`
El sitio de documentación de Sun para todas las tecnologías de Java.

`developers.sun.com`
La página de inicio de Sun para los desarrolladores de Java proporciona descargas, APIs, ejemplos de código, artículos con asesoría técnica y otros recursos sobre las mejores prácticas de desarrollo en Java.

Editores y Entornos de Desarrollo Integrados

`www.eclipse.org`
El entorno de desarrollo Eclipse puede usarse para desarrollar código en cualquier lenguaje de programación. Puede descargar el entorno y varios complementos (plug-ins) de Java para desarrollar sus programas en Java.

`www.netbeans.org`
El IDE NetBeans. Una de las herramientas de desarrollo para Java más populares, de distribución gratuita.

`borland.com/products/downloads/download_jbuilder.html`
Borland ofrece una versión Foundation Edition gratuita de su popular IDE JBuilder para Java. Este sitio también ofrece versiones de prueba de 30 días de las ediciones Enterprise y Developer.

`www.blueJ.org`
BlueJ: una herramienta gratuita diseñada para ayudar a enseñar Java orientado a objetos a los programadores novatos.

`www.jgrasp.org`
Descargas, documentación y tutoriales sobre jGRASP. Esta herramienta muestra representaciones visuales de programas en Java, para ayudar a su comprensión.

`www.jedit.org`
jEdit: un editor de texto escrito en Java.

`developers.sun.com/prodtech/javatools/jsenterprise/index.jsp`
El IDE Sun Java Studio Enterprise: la versión mejorada de NetBeans de Sun Microsystems.

`www.jcreator.com`
JCreator: un IDE popular para Java. JCreator Lite Edition está disponible como descarga gratuita. También está disponible una versión de prueba de 30 días de JCreator Pro Edition.

`www.textpad.com`
TextPad: compile, edite y ejecute sus programas en Java desde este editor, que proporciona coloreo de sintaxis y una interfaz fácil de usar.

`www.download.com`
Un sitio que contiene descargas de aplicaciones de freeware y shareware, incluyendo programas editores.

Sitios de recursos adicionales sobre Java

`www.javalobby.org`
Proporciona noticias actualizadas sobre Java, foros en donde los desarrolladores pueden intercambiar tips y consejos, y una base de conocimiento de Java extensa, que organiza artículos y descargas en toda la Web.

`www.jguru.com`
Ofrece foros, descargas, artículos, cursos en línea y una extensa colección de FAQs (Preguntas frecuentes) sobre Java.

`www.javaworld.com`
Ofrece recursos para desarrolladores de Java, como artículos, índices de libros populares sobre Java, tips y FAQs.

`www.ftponline.com/javapro`
La revista JavaPro contiene artículos mensuales, tips de programación, reseñas de libros y mucho más.

`sys-con.com/java/`
El Diario de Desarrolladores de Java de Sys-Con Media ofrece artículos, libros electrónicos y otros recursos sobre Java.

Resumen

Sección 1.1 Introducción

- Java se ha convertido en el lenguaje de elección para implementar aplicaciones basadas en Internet y software para dispositivos que se comunican a través de una red.
- Java Enterprise Edition (Java EE) está orientada hacia el desarrollo de aplicaciones de redes distribuidas de gran escala, y aplicaciones basadas en Web.
- Java Micro Edition (Java ME) está orientada hacia el desarrollo de aplicaciones para dispositivos pequeños, con memoria limitada, como teléfonos celulares, radiolocalizadotes y PDAs.

Sección 1.2 ¿Qué es una computadora?

- Una computadora es un dispositivo capaz de realizar cálculos y tomar decisiones lógicas a velocidades de millones (incluso de miles de millones) de veces más rápidas que los humanos.
- Las computadoras procesan los datos bajo el control de conjuntos de instrucciones llamadas programas de cómputo. Los programas guían a las computadoras a través de acciones especificadas por gente llamada programadores de computadoras.
- Una computadora está compuesta por varios dispositivos conocidos como hardware. A los programas que se ejecutan en una computadora se les denomina software.

Sección 1.3 Organización de una computadora

- Casi todas las computadoras pueden representarse mediante seis unidades lógicas o secciones.
- La unidad de entrada obtiene información desde los dispositivos de entrada y pone esta información a disposición de las otras unidades para que pueda procesarse.
- La unidad de salida toma información que ya ha sido procesada por la computadora y la coloca en los diferentes dispositivos de salida, para que esté disponible fuera de la computadora.
- La unidad de memoria es la sección de "almacén" de acceso rápido, pero con relativa baja capacidad, de la computadora. Retiene la información que se introduce a través de la unidad de entrada, para que la información pueda estar disponible de manera inmediata para procesarla cuando sea necesario. También retiene la información procesada hasta que ésta pueda ser colocada en los dispositivos de salida por la unidad de salida.
- La unidad aritmética y lógica (ALU) es la responsable de realizar cálculos (como suma, resta, multiplicación y división) y tomar decisiones.

- La unidad central de procesamiento (CPU) coordina y supervisa la operación de las demás secciones. La CPU le indica a la unidad de entrada cuándo debe grabarse la información dentro de la unidad de memoria, a la ALU cuándo debe utilizarse la información de la unidad de memoria para los cálculos, y a la unidad de salida cuándo enviar la información desde la unidad de memoria hasta ciertos dispositivos de salida.
- Los multiprocesadores contienen múltiples CPUs y, por lo tanto, pueden realizar muchas operaciones de manera simultánea.
- La unidad de almacenamiento secundario es la sección de "almacén" de alta capacidad y de larga duración de la computadora. Los programas o datos que no se encuentran en ejecución por las otras unidades, normalmente se colocan en dispositivos de almacenamiento secundario hasta que son requeridos de nuevo.

Sección 1.4 Los primeros sistemas operativos

- Las primeras computadoras eran capaces de realizar solamente una tarea o trabajo a la vez.
- Los sistemas operativos se desarrollaron para facilitar el uso de la computadora.
- La multiprogramación significa la operación simultánea de muchas tareas.
- Con el tiempo compartido, la computadora ejecuta una pequeña porción del trabajo de un usuario y después procede a dar servicio al siguiente usuario, con la posibilidad de proporcionar el servicio a cada usuario varias veces por segundo.

Sección 1.5 Computación personal, distribuida y cliente/servidor

- En 1977, Apple Computer popularizó el fenómeno de la computación personal.
- En 1981, IBM, el vendedor de computadoras más grande del mundo, introdujo la Computadora Personal (PC) de IBM, que legitimó rápidamente la computación en las empresas, en la industria y en las organizaciones gubernamentales.
- En la computación distribuida, en vez de que la computación se realice sólo en una computadora central, se distribuye mediante redes a los sitios en donde se realiza el trabajo de la empresa.
- Los servidores almacenan datos que pueden utilizar las computadoras cliente distribuidas a través de la red, de ahí el término de computación cliente/servidor.
- Java se está utilizando ampliamente para escribir software para redes de computadoras y para aplicaciones cliente/servidor distribuidas.

Sección 1.6 Internet y World Wide Web

- Internet es accesible por más de mil millones de computadoras y dispositivos controlados por computadora.
- Con la introducción de World Wide Web, Internet se ha convertido explosivamente en uno de los principales mecanismos de comunicación en todo el mundo.

Sección 1.7 Lenguajes máquina, lenguajes ensambladores y lenguajes de alto nivel

- Cualquier computadora puede entender de manera directa sólo su propio lenguaje máquina.
- El lenguaje máquina es el "lenguaje natural" de una computadora.
- Por lo general, los lenguajes máquina consisten en cadenas de números (que finalmente se reducen a 1s y 0s) que instruyen a las computadoras para realizar sus operaciones más elementales, una a la vez.
- Los lenguajes máquina son dependientes de la máquina.
- Los programadores empezaron a utilizar abreviaturas del inglés para representar las operaciones elementales. Estas abreviaturas formaron la base de los lenguajes ensambladores.
- Los programas traductores conocidos como ensambladores se desarrollaron para convertir los primeros programas en lenguaje ensamblador a lenguaje máquina, a la velocidad de la computadora.
- Los lenguajes de alto nivel permiten a los programadores escribir instrucciones parecidas al lenguaje inglés cotidiano, y contienen notaciones matemáticas de uso común.
- Java es el lenguaje de programación de alto nivel más utilizado en todo el mundo.
- Los programas intérpretes ejecutan los programas en lenguajes de alto nivel directamente.

Sección 1.8 Historia de C y C++

- Java evolucionó de C++, el cual evolucionó de C, que a su vez evolucionó de BCPL y B.
- El lenguaje C evolucionó a partir de B, gracias al trabajo de Dennis Ritchie en los laboratorios Bell. Inicialmente, se hizo muy popular como lenguaje de desarrollo para el sistema operativo UNIX.
- A principios de la década de los ochenta, Bjarne Stroustrup desarrolló una extensión de C en los laboratorios Bell: C++. Este lenguaje proporciona un conjunto de características que "pulen" al lenguaje C, además de la capacidad de una programación orientada a objetos.

Sección 1.9 Historia de Java

- Java se utiliza para desarrollar aplicaciones empresariales a gran escala, para mejorar la funcionalidad de los servidores Web, para proporcionar aplicaciones para los dispositivos domésticos y para muchos otros propósitos.
- Los programas en Java consisten en piezas llamadas clases. Las clases incluyen piezas llamadas métodos, los cuales realizan tareas y devuelven información cuando se completan estas tareas.

Sección 1.10 Bibliotecas de clases de Java

- La mayoría de los programadores en Java aprovechan las ricas colecciones de clases existentes en las bibliotecas de clases de Java, que también se conocen como APIs (Interfaces de programación de aplicaciones) de Java.
- La ventaja de crear sus propias clases y métodos es que sabe cómo funcionan y puede examinar el código. La desventaja es que se requiere una cantidad considerable de tiempo y un esfuerzo potencialmente complejo.

Sección 1.11 FORTRAN, COBOL, Pascal y Ada

- Fortran (FORmula TRANslator, Traductor de fórmulas) fue desarrollado por IBM Corporation a mediados de la década de los cincuenta para utilizarse en aplicaciones científicas y de ingeniería que requerían cálculos matemáticos complejos.
- COBOL (COmmon Business Oriented Language, Lenguaje común orientado a negocios) se utiliza en aplicaciones comerciales que requieren de una manipulación precisa y eficiente de grandes volúmenes de datos.
- Las actividades de investigación en la década de los sesenta dieron como resultado la evolución de la programación estructurada (un método disciplinado para escribir programas que sean más claros, fáciles de probar y depurar, y más fáciles de modificar que los programas extensos producidos con técnicas anteriores).
- Pascal se diseñó para la enseñanza de la programación estructurada en ambientes académicos, y de inmediato se convirtió en el lenguaje de programación preferido en la mayoría de las universidades.
- El lenguaje de programación Ada se desarrolló bajo el patrocinio del Departamento de Defensa de los Estados Unidos (DOD) para satisfacer la mayoría de sus necesidades. Una característica de Ada conocida como multitarea permite a los programadores especificar que muchas actividades ocurrirán en paralelo. Java, a través de una técnica que se conoce como *subprocesamiento múltiple*, también permite a los programadores escribir programas con actividades paralelas.

Sección 1.12 BASIC, Visual Basic, Visual C++, C# y .NET

- BASIC fue desarrollado a mediados de la década de los sesenta para escribir programas simples.
- El lenguaje Visual Basic de Microsoft simplifica el desarrollo de aplicaciones para Windows.
- La plataforma .NET de Microsoft integra Internet y Web en las aplicaciones de computadora.

Sección 1.13 Entorno de desarrollo típico en Java

- Por lo general, los programas en Java pasan a través de cinco fases: edición, compilación, carga, verificación y ejecución.
- La fase 1 consiste en editar un archivo con un editor. Usted escribe un programa utilizando el editor, realiza las correcciones necesarias y guarda el programa en un dispositivo de almacenamiento secundario, tal como su disco duro.
- Un nombre de archivo que termina con la extensión `.java` indica que éste contiene código fuente en Java.
- Los entornos de desarrollo integrados (IDEs) proporcionan herramientas que dan soporte al proceso de desarrollo del software, incluyendo editores para escribir y editar programas, y depuradores para localizar errores lógicos.
- En la fase 2, el programador utiliza el comando `javac` para compilar un programa.
- Si un programa se compila, el compilador produce un archivo `.class` que contiene el programa compilado.
- El compilador de Java traduce el código fuente de Java en códigos de bytes que representan las tareas a ejecutar. La Máquina Virtual de Java (JVM) ejecuta los códigos de bytes.
- En la fase 3, de carga, el cargador de clases toma los archivos `.class` que contienen los códigos de bytes del programa y los transfiere a la memoria principal.
- En la fase 4, a medida que se cargan las clases, el verificador de códigos de bytes examina sus códigos de bytes para asegurar que sean válidos y que no violen las restricciones de seguridad de Java.
- En la fase 5, la JVM ejecuta los códigos de bytes del programa.

Sección 1.16 Ejemplo práctico de Ingeniería de Software: introducción a la tecnología de objetos y UML

- El Lenguaje Unificado de Modelado (UML) es un lenguaje gráfico que permite a las personas que crean sistemas representar sus diseños orientados a objetos en una notación común.

- El diseño orientado a objetos (DOO) modela los componentes de software en términos de objetos reales.
- Los objetos tienen la propiedad de ocultamiento de la información: por lo general, no se permite a los objetos de una clase saber cómo se implementan los objetos de otras clases.
- La programación orientada a objetos (POO) implementa diseños orientados a objetos.
- Los programadores de Java se concentran en crear sus propios tipos definidos por el usuario, conocidos como clases. Cada clase contiene datos y métodos que manipulan a esos datos y proporcionan servicios a los clientes.
- Los componentes de datos de una clase son los atributos o campos; los componentes de operación son los métodos.
- Las clases pueden tener relaciones con otras clases; a estas relaciones se les llama asociaciones.
- El proceso de empaquetar software en forma de clases hace posible que los sistemas de software posteriores reutilicen esas clases.
- A una instancia de una clase se le llama objeto.
- El proceso de analizar y diseñar un sistema desde un punto de vista orientado a objetos se llama análisis y diseño orientados a objetos (A/DOO).

Terminología

Ada
ALU (unidad aritmética y lógica)
ANSI C
API de Java (Interfaz de Programación de Aplicaciones)
atributo
BASIC
bibliotecas de clases
C
C#
C++
cargador de clases
clase
`.class`, archivo
COBOL
código de bytes
compilador
compilador HotSpot™
compilador JIT (justo a tiempo)
componente reutilizable
comportamiento
computación cliente/servidor
computación distribuida
computación personal
computadora
contenido dinámico
CPU (unidad central de procesamiento)
disco
diseño orientado a objetos (DOO)
dispositivo de entrada
dispositivo de salida
documento de requerimientos
editor
encapsulamiento
ensamblador
entrada/salida (E/S)
enunciado del problema
error en tiempo de compilación
error en tiempo de ejecución
error fatal en tiempo de ejecución
error no fatal en tiempo de ejecución
fase de carga
fase de compilación
fase de edición
fase de ejecución
fase de verificación
flujo de datos
Fortran
Hardware
herencia
HTML (Lenguaje de Marcado de Hipertexto)
IDE (Entorno Integrado de Desarrollo)
Internet
Intérprete
Java
Java Enterprise Edition (Java EE)
`.java`, extensión de nombre de archivo
Java Micro Edition (Java ME)
Java Standard Edition (Java SE)
`java`, intérprete
`javac`, compilador
KIS (simplifíquelo)
Kit de Desarrollo de Java (JDK)
LAN (red de área local)
lenguaje de alto nivel
lenguaje ensamblador
lenguaje máquina
Lenguaje Unificado de Modelado (UML)
Máquina virtual de Java (JVM)
memoria principal
método
método de código activo (live-code)
Microsoft Internet Explorer, navegador Web
modelado
multiprocesador
multiprogramación
.NET
objeto
ocultamiento de información

Pascal
plataforma
portabilidad
programa de cómputo
programa traductor
programación estructurada
programación orientada a objetos (POO)
programación por procedimientos
programador de computadoras
pseudocódigo
reutilización de software
servidor de archivos
sistema heredado
sistema operativo
software
subprocesamiento múltiple
Sun Microsystems
tiempo compartido
tipo definido por el usuario
traducción
unidad aritmética y lógica (ALU)
unidad central de procesamiento (CPU)
unidad de almacenamiento secundario
unidad de entrada
unidad de memoria
unidad de salida
verificador de código de bytes
Visual Basic .NET
Visual C++ .NET
World Wide Web

Ejercicios de autoevaluación

1.1 Complete las siguientes oraciones:

a) La compañía que popularizó la computación personal fue ______________.
b) La computadora que legitimó la computación personal en los negocios y la industria fue ______________.
c) Las computadoras procesan los datos bajo el control de conjuntos de instrucciones llamadas ____________.
d) Las seis unidades lógicas clave de la computadora son ______________, ______________, ______________, ______________, ______________ y ______________.
e) Los tres tipos de lenguajes descritos en este capítulo son ______________, ______________ y ______________.
f) Los programas que traducen programas en lenguaje de alto nivel a lenguaje máquina se denominan ______________.
g) La ___________ permite a los usuarios de computadora localizar y ver documentos basados en multimedia sobre casi cualquier tema, a través de Internet.
h) ______________, permite a un programa en Java realizar varias actividades en paralelo.

1.2 Complete cada una de las siguientes oraciones relacionadas con el entorno de Java:

a) El comando ______________ del JDK ejecuta una aplicación en Java.
b) El comando ______________ del JDK compila un programa en Java.
c) El archivo de un programa en Java debe terminar con la extensión de archivo ______________.
d) Cuando se compila un programa en Java, el archivo producido por el compilador termina con la extensión ______________.
e) El archivo producido por el compilador de Java contiene ______________ que se ejecutan mediante la Máquina Virtual de Java.

1.3 Complete cada una de las siguientes oraciones (basándose en la sección 1.16):

a) Los objetos tienen una propiedad que se conoce como ______________; aunque éstos pueden saber cómo comunicarse con los demás objetos a través de interfaces bien definidas, generalmente no se les permite saber cómo están implementados los otros objetos.
b) Los programadores de Java se concentran en crear ______________, que contienen campos y el conjunto de métodos que manipulan a esos campos y proporcionan servicios a los clientes.
c) Las clases pueden tener relaciones con otras clases; a éstas relaciones se les llama ______________.
d) El proceso de analizar y diseñar un sistema desde un punto de vista orientado a objetos se conoce como ______________ .
e) El DOO aprovecha las relaciones ______________, en donde se derivan nuevas clases de objetos al absorber las características de las clases existentes y después agregar sus propias características únicas.
f) ______________ es un lenguaje gráfico que permite a las personas que diseñan sistemas de software utilizar una notación estándar en la industria para representarlos.
g) El tamaño, forma, color y peso de un objeto se consideran ______________ del mismo.

Respuestas a los ejercicios de autoevaluación

1.1 a) Apple. b) PC de IBM. c) programas. d) unidad de entrada, unidad de salida, unidad de memoria, unidad aritmética y lógica, unidad central de procesamiento, unidad de almacenamiento secundario. e) lenguajes máquina, lenguajes ensambladores, lenguajes de alto nivel. f) compiladores. g) World Wide Web. h) Subprocesamiento múltiple.

1.2 a) `java.` b) `javac.` c) `.java.` d) `.class.` e) códigos de bytes.

1.3 a) ocultamiento de información. b) clases. c) asociaciones. d) análisis y diseño orientados a objetos (A/DOO). e) herencia. f) El Lenguaje Unificado de Modelado (UML). g) atributos.

Ejercicios

1.4 Clasifique cada uno de los siguientes elementos como hardware o software:
- a) CPU
- b) compilador de Java
- c) JVM
- d) unidad de entrada
- e) editor

1.5 Complete cada una de las siguientes oraciones:
- a) La unidad lógica de la computadora que recibe información desde el exterior de la computadora para que ésta la utilice se llama ______________.
- b) El proceso de indicar a la computadora cómo resolver problemas específicos se llama ______________.
- c) ______________ es un tipo de lenguaje computacional que utiliza abreviaturas del inglés para las instrucciones de lenguaje máquina.
- d) ______________ es una unidad lógica de la computadora que envía información, que ya ha sido procesada por la computadora, a varios dispositivos, de manera que la información pueda utilizarse fuera de la computadora.
- e) ______________ y ______________ son unidades lógicas de la computadora que retienen información.
- f) ______________ es una unidad lógica de la computadora que realiza cálculos.
- g) ______________ es una unidad lógica de la computadora que toma decisiones lógicas.
- h) Los lenguajes ______________ son los más convenientes para que el programador pueda escribir programas rápida y fácilmente.
- i) Al único lenguaje que una computadora puede entender directamente se le conoce como el ___________ de esa computadora.
- j) ______________ es una unidad lógica de la computadora que coordina las actividades de todas las demás unidades lógicas.

1.6 Indique la diferencia entre los términos error fatal y error no fatal. ¿Por qué sería preferible experimentar un error fatal, en vez de un error no fatal?

1.7 Complete cada una de las siguientes oraciones:
- a) ______________ se utiliza ahora para desarrollar aplicaciones empresariales de gran escala, para mejorar la funcionalidad de los servidores Web, para proporcionar aplicaciones para dispositivos domésticos y para muchos otros fines más.
- b) ______________ se diseñó específicamente para la plataforma .NET, de manera que los programadores pudieran migrar fácilmente a .NET.
- c) Inicialmente, ______________ se hizo muy popular como lenguaje de desarrollo para el sistema operativo UNIX.
- d) ______________ fue desarrollado a mediados de la década de los sesenta en el Dartmouth College, como un medio para escribir programas simples.
- e) ______________ fue desarrollado por IBM Corporation a mediados de la década de los cincuenta para utilizarse en aplicaciones científicas y de ingeniería que requerían cálculos matemáticos complejos.
- f) ______________ se utiliza para aplicaciones comerciales que requieren la manipulación precisa y eficiente de grandes cantidades de datos.

g) El lenguaje de programación ____________ fue desarrollado por Bjarne Stroustrup a principios de la década de los ochenta, en los laboratorios Bell.

1.8 Complete cada una de las siguientes oraciones (basándose en la sección 1.13):

a) Por lo general, los programas de Java pasan a través de cinco fases: ____________, ____________, ____________, ____________ y ____________.

b) Un ____________ proporciona muchas herramientas que dan soporte al proceso de desarrollo de software, como los editores para escribir y editar programas, los depuradores para localizar los errores lógicos en los programas, y muchas otras características más.

c) El comando `java` invoca al ____________, que ejecuta los programas de Java.

d) Un(a) ____________ es una aplicación de software que simula a una computadora, pero oculta el sistema operativo subyacente y el hardware de los programas que interactúan con la VM.

e) Un programa ____________ puede ejecutarse en múltiples plataformas.

f) El ____________ toma los archivos `.class` que contienen los códigos de bytes del programa y los transfiere a la memoria principal.

g) El ____________ examina los códigos de bytes para asegurar que sean válidos.

1.9 Explique las dos fases de compilación de los programas de Java.

2

Introducción a las aplicaciones en Java



¿Qué hay en un nombre? A eso a lo que llamamos rosa, si le diéramos otro nombre conservaría su misma fragancia dulce.

—William Shakespeare

Al hacer frente a una decisión, siempre me pregunto, "¿Cuál será la solución más divertida?"

—Peggy Walker

"Toma un poco más de té", dijo el conejo blanco a Alicia, con gran seriedad. "No he tomado nada todavía." Contestó Alicia en tono ofendido, "Entonces no puedo tomar más". "Querrás decir que no puedes tomar menos", dijo el sombrerero loco, "es muy fácil tomar más que nada".

—Lewis Carroll

OBJETIVOS

En este capítulo aprenderá a:

- Escribir aplicaciones simples en Java.
- Utilizar las instrucciones de entrada y salida.
- Familiarizarse con los tipos primitivos de Java.
- Comprender los conceptos básicos de la memoria.
- Utilizar los operadores aritméticos.
- Comprender la precedencia de los operadores aritméticos.
- Escribir instrucciones para tomar decisiones.
- Utilizar los operadores relacionales y de igualdad.

Plan general

2.1 Introducción

Ahora presentaremos la programación de aplicaciones en Java, que facilita una metodología disciplinada para el diseño de programas. La mayoría de los programas en Java que estudiará en este libro procesan información y muestran resultados. Le presentaremos seis ejemplos que demuestran cómo sus programas pueden mostrar mensajes y cómo pueden obtener información del usuario para procesarla. Comenzaremos con varios ejemplos que simplemente muestran mensajes en la pantalla. Después demostraremos un programa que obtiene dos números de un usuario, calcula su suma y muestra el resultado. Usted aprenderá a realizar varios cálculos aritméticos y a guardar sus resultados para usarlos más adelante. El último ejemplo en este capítulo demuestra los fundamentos de toma de decisiones, al mostrarle cómo comparar números y después mostrar mensajes con base en los resultados de la comparación. Por ejemplo, el programa muestra un mensaje que indica que dos números son iguales sólo si tienen el mismo valor. Analizaremos cada ejemplo, una línea a la vez, para ayudarle a aprender a programar en Java. En los ejercicios del capítulo proporcionamos muchos problemas retadores y divertidos, para ayudarle a aplicar las habilidades que aprenderá aquí.

2.2 Su primer programa en Java: imprimir una línea de texto

Cada vez que utiliza una computadora, ejecuta diversas aplicaciones que realizan tareas por usted. Por ejemplo, su aplicación de correo electrónico le permite enviar y recibir mensajes de correo, y su navegador Web le permite ver páginas de sitios Web en todo el mundo. Los programadores de computadoras crean dichas aplicaciones, escribiendo **programas de cómputo**.

Una **aplicación** en Java es un programa de computadora que se ejecuta cuando usted utiliza el comando `java` para iniciar la Máquina Virtual de Java (JVM). Consideremos una aplicación simple que muestra una línea de texto. (Más adelante en esta sección hablaremos sobre cómo compilar y ejecutar una aplicación). El programa y su salida se muestran en la figura 2.1. La salida aparece en el recuadro al final del programa. El programa ilustra varias características importantes del lenguaje. Java utiliza notaciones que pueden parecer extrañas a los no programadores. Además, cada uno de los programas que presentamos en este libro tiene números de línea incluidos para su conveniencia; los números de línea no son parte de los programas en Java. Pronto veremos que la línea 9 se encarga del verdadero trabajo del programa; a saber, mostrar la frase `Bienvenido a la programacion en Java!` en la pantalla. Ahora consideremos cada línea del programa en orden.

La línea 1

```
// Fig. 2.1: Bienvenido1.java
```

empieza con `//`, indicando que el resto de la línea es un **comentario**. Los programadores insertan comentarios para **documentar los programas** y mejorar su legibilidad. Los comentarios también ayudan a otras personas a leer y comprender un programa. El compilador de Java ignora estos comentarios, de manera que la computadora no hace nada cuando el programa se ejecuta. Por convención, comenzamos cada uno de los programas con un comentario, el cual indica el número de figura y el nombre del archivo.

```
1   // Fig. 2.1: Bienvenido1.java
2   // Programa para imprimir texto.
3
4   public class Bienvenido1
5   {
6      // el método main empieza la ejecución de la aplicación en Java
7      public static void main( String args[] )
8      {
9         System.out.println( "Bienvenido a la programacion en Java!" );
10
11     } // fin del método main
12
13  } // fin de la clase Bienvenido1
```

```
Bienvenido a la programacion en Java!
```

Figura 2.1 | Programa para imprimir texto.

Un comentario que comienza con `//` se llama **comentario de fin de línea** (o **de una sola línea**), ya que termina al final de la línea en la que aparece. Un comentario que se especifica con `//` puede empezar también en medio de una línea, y continuar solamente hasta el final de esa línea (como en las líneas 11 y 13).

Los **comentarios tradicionales** (también conocidos como **comentarios de múltiples líneas**), como el que se muestra a continuación

```
/* Éste es un comentario
   Tradicional. Puede
   dividirse en muchas líneas */
```

se distribuyen en varias líneas. Este tipo de comentario comienza con el delimitador `/*` y termina con `*/`. El compilador ignora todo el texto que esté entre los delimitadores. Java incorporó los comentarios tradicionales y los comentarios de fin de línea de los lenguajes de programación C y C++, respectivamente. En este libro utilizamos comentarios de fin de línea.

Java también cuenta con **comentarios Javadoc**, que están delimitados por `/**` y `*/`. Al igual que con los comentarios tradicionales, el compilador ignora todo el texto entre los delimitadores de los comentarios Javadoc. Estos comentarios permiten a los programadores incrustar la documentación del programa directamente en éste. Dichos comentarios son el formato preferido en la industria. El programa de utilería **`javadoc`** (parte del Kit de Desarrollo de Java SE) lee esos comentarios y los utiliza para preparar la documentación de su programa, en formato HTML. Hay algunas sutilezas en cuanto al uso apropiado de los comentarios estilo Java. En el apéndice K, Creación de documentación con `javadoc`, demostramos el uso de los comentarios Javadoc y la herramienta `javadoc`. Para obtener información completa, visite la página de herramientas de `javadoc` de Sun en `java.sun.com/javase/6/docs/technotes/guides/javadoc/index.html`.



Error común de programación 2.1

Olvidar uno de los delimitadores de un comentario tradicional o Javadoc es un error de sintaxis. La ***sintaxis*** *de un lenguaje de programación que especifica las reglas para crear un programa apropiado en ese lenguaje. Un* ***error de sintaxis*** *ocurre cuando el compilador encuentra código que viola las reglas del lenguaje Java (es decir, su sintaxis). En este caso, el compilador muestra un mensaje de error para ayudar al programador a identificar y corregir el código incorrecto. Los errores de sintaxis se conocen también como* ***errores del compilador****,* ***errores en tiempo de compilación*** *o errores de compilación, ya que el compilador los detecta durante la fase de compilación. Usted no podrá ejecutar su programa sino hasta que corrija todos los errores de sintaxis que éste contenga.*

La línea 2

```
// Programa para imprimir texto.
```

es un comentario de fin de línea que describe el propósito del programa.



Buena práctica de programación 2.1

Es conveniente que todo programa comience con un comentario que explique su propósito, el autor, la fecha y la hora de la última modificación del mismo. (No mostraremos el autor, la fecha y la hora en los programas de este libro, ya que sería redundante).

La línea 3 es una línea en blanco. Los programadores usan líneas en blanco y espacios para facilitar la lectura de los programas. En conjunto, las líneas en blanco, los espacios y los tabuladores se conocen como **espacio en blanco**. (Los espacios y tabuladores se conocen específicamente como **caracteres de espacio en blanco**). El compilador ignora el espacio en blanco. En éste y en los siguientes capítulos, hablaremos sobre las convenciones para utilizar espacios en blanco para mejorar la legibilidad de los programas.



Buena práctica de programación 2.2

Utilice líneas en blanco y espacios para mejorar la legibilidad del programa.

La línea 4

```
public class Bienvenido1
```

comienza una **declaración de clase** para la clase `Bienvenido1`. Todo programa en Java consiste de, cuando menos, una declaración de clase que usted, el programador, debe definir. Estas clases se conocen como **clases definidas por el programador** o **clases definidas por el usuario**. La **palabra clave `class`** introduce una declaración de clase en Java, la cual debe ir seguida inmediatamente por el **nombre de la clase** (`Bienvenido1`). Las palabras clave (algunas veces conocidas como **palabras reservadas**) se reservan para uso exclusivo de Java (hablaremos sobre las diversas palabras clave a lo largo de este texto) y siempre se escriben en minúscula. En el apéndice C se muestra la lista completa de palabras clave de Java.

Por convención, todos los nombres de clases en Java comienzan con una letra mayúscula, y la primera letra de cada palabra en el nombre de la clase debe ir en mayúscula (por ejemplo, `EjemploDeNombreDeClase`). En Java, el nombre de una clase se conoce como **identificador**: una serie de caracteres que pueden ser letras, dígitos, guiones bajos (`_`) y signos de moneda (`$`), que no comience con un dígito ni tenga espacios. Algunos identificadores válidos son `Bienvenido1`, `$valor`, `_valor`, `m_campoEntrada1` y `boton7`. El nombre `7boton` no es un identificador válido, ya que comienza con un dígito, y el nombre `campo entrada` tampoco lo es debido a que contiene un espacio. Por lo general, un identificador que no empieza con una letra mayúscula no es el nombre de una clase. Java es **sensible a mayúsculas y minúsculas**; es decir, las letras mayúsculas y minúsculas son distintas, por lo que `a1` y `A1` son distintos identificadores (pero ambos son válidos).



Buena práctica de programación 2.3

Por convención, el identificador del nombre de una clase siempre debe comenzar con una letra mayúscula, y la primera letra de cada palabra subsiguiente del identificador también debe ir en mayúscula. Los programadores de Java saben que, por lo general, dichos identificadores representan clases de Java, por lo que si usted nombra a sus clases de esta forma, sus programas serán más legibles.



Error común de programación 2.2

Java es sensible a mayúsculas y minúsculas. No utilizar la combinación apropiada de letras minúsculas y mayúsculas para un identificador, generalmente produce un error de compilación.

En los capítulos 2 al 7, cada una de las clases que definimos comienza con la palabra clave **`public`**. Cuando usted guarda su declaración de clase `public` en un archivo, el nombre del mismo debe ser el nombre de la clase, seguido de la **extensión de nombre de archivo `.java`**. Para nuestra aplicación, el nombre del archivo es `Bienvenido1.java`. En el capítulo 8 aprenderá más acerca de las clases `public` y las que no son `public`.



Error común de programación 2.3

Una clase `public` debe colocarse en un archivo que tenga el mismo nombre que la clase (en términos de ortografía y uso de mayúsculas) y la extensión `.java`; en caso contrario, ocurre un error de compilación.



Error común de programación 2.4

Es un error que un archivo que contiene la declaración de una clase, no finalice con la extensión `.java`. *El compilador de Java sólo compila archivos con la extensión* `.java`.

Una **llave izquierda** (en la línea 5 de este programa), `{`, comienza el **cuerpo** de todas las declaraciones de clases. Su correspondiente **llave derecha** (en la línea 13), `}`, debe terminar cada declaración de una clase. Observe que las líneas de la 6 a la 11 tienen sangría; ésta es una de las convenciones de espaciado que se mencionaron anteriormente. Definimos cada una de las convenciones de espaciado como una Buena práctica de programación.



Buena práctica de programación 2.4

Siempre que escriba una llave izquierda de apertura (`{`) en su programa, escriba inmediatamente la llave derecha de cierre (`}`) y luego vuelva a colocar el cursor entre las llaves y utilice sangría para comenzar a escribir el cuerpo. Esta práctica ayuda a evitar errores debido a la omisión de una de las llaves.



Buena práctica de programación 2.5

Aplique sangría a todo el cuerpo de la declaración de cada clase, usando un "nivel" de sangría entre la llave izquierda (`{`) y la llave derecha (`}`), las cuales delimitan el cuerpo de la clase. Este formato enfatiza la estructura de la declaración de la clase, y facilita su lectura.



Buena práctica de programación 2.6

Establezca una convención para el tamaño de sangría que usted prefiera, y después aplique uniformemente esta convención. La tecla **Tab** *puede utilizarse para crear sangrías, pero las posiciones de los tabuladores pueden variar entre los diversos editores de texto. Le recomendamos utilizar tres espacios para formar un nivel de sangría.*



Error común de programación 2.5

Es un error de sintaxis no utilizar las llaves por pares.

La línea 6

```
// el método main empieza la ejecución de la aplicación en Java
```

es un comentario de fin de línea que indica el propósito de las líneas 7 a 11 del programa. La línea 7

```
public static void main( String args[] )
```

es el punto de inicio de toda aplicación en Java. Los **paréntesis** después del identificador `main` indican que éste es un bloque de construcción del programa, al cual se le llama **método**. Las declaraciones de clases en Java generalmente contienen uno o más métodos. En una aplicación en Java, sólo uno de esos métodos debe llamarse `main` y debe definirse como se muestra en la línea 7; de no ser así, la JVM no ejecutará la aplicación. Los métodos pueden realizar tareas y devolver información una vez que las hayan concluido. La palabra clave **`void`** indica que este método realizará una tarea, pero no devolverá ningún tipo de información cuando complete su tarea. Más adelante veremos que muchos métodos devuelven información cuando finalizan sus tareas. Aprenderá más acerca de los métodos en los capítulos 3 y 6. Por ahora, simplemente copie la primera línea de `main` en sus aplicaciones en Java. En la línea 7, las palabras `String args[]` entre paréntesis son una parte requerida de la declaración del método `main`. Hablaremos sobre esto en el capítulo 7, Arreglos.

La llave izquierda (`{`) en la línea 8 comienza el **cuerpo de la declaración del método**; su correspondiente llave derecha (`}`) debe terminar el cuerpo de esa declaración (línea 11 del programa). Observe que la línea 9, entre las llaves, tiene sangría.



Buena práctica de programación 2.7

Aplique un "nivel" de sangría a todo el cuerpo de la declaración de cada método, entre la llave izquierda (`{`) y la llave derecha (`}`), las cuales delimitan el cuerpo del método. Este formato resalta la estructura del método y ayuda a que su declaración sea más fácil de leer.

La línea 9

```
System.out.println( "Bienvenido a la programacion en Java!" );
```

indica a la computadora que **realice una acción**; es decir, que imprima la **cadena** de caracteres contenida entre los caracteres de comillas dobles (sin incluirlas). A una cadena también se le denomina **cadena de caracteres**, **mensaje** o **literal de cadena**. Genéricamente, nos referimos a los caracteres entre comillas dobles como **cadenas**. El compilador no ignora los caracteres de espacio en blanco dentro de las cadenas.

`System.out` se conoce como el **objeto de salida estándar**. `System.out` permite a las aplicaciones en Java mostrar conjuntos de caracteres en la **ventana de comandos**, desde la cual se ejecuta la aplicación en Java. En Microsoft Windows 95/98/ME, la ventana de comandos es el **símbolo de MS-DOS**. En versiones más recientes de Microsoft Windows, la ventana de comandos es el **Símbolo del sistema**. En UNIX/Linux/Mac OS X, la ventana de comandos se llama **ventana de terminal** o **shell**. Muchos programadores se refieren a la ventana de comandos simplemente como la **línea de comandos**.

El método **`System.out.println` muestra** (o **imprime**) **una línea** de texto en la ventana de comandos. La cadena dentro de los paréntesis en la línea 9 es el **argumento** para el método. El método `System.out.println` realiza su tarea, mostrando (o enviando) su argumento en la ventana de comandos. Cuando `System.out.println` completa su tarea, posiciona el **cursor de salida** (la ubicación en donde se mostrará el siguiente carácter) al principio de la siguiente línea en la ventana de comandos. [Este desplazamiento del cursor es similar a cuando un usuario oprime la tecla *Intro*, al escribir en un editor de texto (el cursor aparece al principio de la siguiente línea en el archivo)].

Toda la línea 9, incluyendo `System.out.println`, el argumento `"Bienvenido a la programacion en Java!"` entre paréntesis y el **punto y coma** (`;`), se conoce como una **instrucción**; y siempre debe terminar con un punto y coma. Cuando se ejecuta la instrucción de la línea 9 de nuestro programa, ésta muestra el mensaje `Bienvenido a la programacion en Java!` en la ventana de comandos. Por lo general, un método está compuesto por una o más instrucciones que realizan la tarea, como veremos en los siguientes programas.



Error común de programación 2.6

Omitir el punto y coma al final de una instrucción es un error de sintaxis.



Tip para prevenir errores 2.1

Al aprender a programar, es conveniente, en ocasiones, "descomponer" un programa funcional, para poder familiarizarse con los mensajes de error de sintaxis del compilador; ya que este tipo de mensajes no siempre indican el problema exacto en el código. Y de esta manera, cuando se encuentren dichos mensajes de error de sintaxis, tendrá una idea de qué fue lo que ocasionó el error. Trate de quitar un punto y coma o una llave del programa de la figura 2.1, y vuelva a compilarlo de manera que pueda ver los mensajes de error generados por esta omisión.



Tip para prevenir errores 2.2

Cuando el compilador reporta un error de sintaxis, éste tal vez no se encuentre en el número de línea indicado por el mensaje. Primero verifique la línea en la que se reportó el error; si esa línea no contiene errores de sintaxis, verifique las líneas anteriores.

A algunos programadores se les dificulta, cuando leen o escriben un programa, relacionar las llaves izquierda y derecha (`{` y `}`) que delimitan el cuerpo de la declaración de una clase o de un método. Por esta razón, incluyen un comentario de fin de línea después de una llave derecha de cierre (`}`) que termina la declaración de un método y que termina la declaración de una clase. Por ejemplo, la línea 11

```
} // fin del método main
```

especifica la llave derecha de cierre (`}`) del método `main`, y la línea 13

```
} // fin de la clase Bienvenido1
```

especifica la llave derecha de cierre (`}`) de la clase `Bienvenido1`. Cada comentario indica el método o la clase que termina con esa llave derecha.



Buena práctica de programación 2.8

Para mejorar la legibilidad de los programas, agregue un comentario de fin de línea después de la llave derecha de cierre (}), que indique a qué método o clase pertenece.

Cómo compilar y ejecutar su primera aplicación en Java

Ahora estamos listos para compilar y ejecutar nuestro programa. Para este propósito, supondremos que usted utiliza el Kit de Desarrollo 6.0 (JDK 6.0) de Java SE de Sun Microsystems. En nuestros centros de recursos en `www.deitel.com/ResourceCenters.html` proporcionamos vínculos a tutoriales que le ayudarán a empezar a trabajar con varias herramientas de desarrollo populares de Java.

Para compilar el programa, abra una ventana de comandos y cambie al directorio en donde está guardado el programa. La mayoría de los sistemas operativos utilizan el comando **cd** para **cambiar directorios**. Por ejemplo,

```
cd c:\ejemplos\cap02\fig02_01
```

cambia al directorio `fig02_01` en Windows. El comando

```
cd ~/ejemplos/cap02/fig02_01
```

cambia al directorio `fig02_01` en UNIX/Linux/Mac OS X.

Para compilar el programa, escriba

```
javac Bienvenido1.java
```

Si el programa no contiene errores de sintaxis, el comando anterior crea un nuevo archivo llamado `Bienvenido1.class` (conocido como el **archivo de clase** para `Bienvenido1`), el cual contiene los códigos de bytes de Java que representan nuestra aplicación. Cuando utilicemos el comando `java` para ejecutar la aplicación, la JVM ejecutará estos códigos de bytes.



Tip para prevenir errores 2.3

Cuando trate de compilar un programa, si recibe un mensaje como "`comando o nombre de archivo incorrecto`", "`javac: comando no encontrado`" o "`'javac ' no se reconoce como un comando interno o externo, programa o archivo por lotes ejecutable`", entonces su instalación del software Java no se completó apropiadamente. Con el JDK, esto indica que la variable de entorno `PATH` del sistema no se estableció apropiadamente. Consulte cuidadosamente las instrucciones de instalación en la sección Antes de empezar de este libro. En algunos sistemas, después de corregir la variable `PATH`, es probable que necesite reiniciar su equipo o abrir una nueva ventana de comandos para que estos ajustes tengan efecto.



Tip para prevenir errores 2.4

Cuando la sintaxis de un programa es incorrecta, el compilador de Java genera mensajes de error de sintaxis; éstos contienen el nombre de archivo y el número de línea en donde ocurrió el error. Por ejemplo, `Bienvenido1.java:6` indica que ocurrió un error en el archivo `Bienvenido1.java` en la línea 6. El resto del mensaje proporciona información acerca del error de sintaxis.



Tip para prevenir errores 2.5

El mensaje de error del compilador "`Public class` NombreClase `must be defined in a file called` NombreClase`.java`" indica que el nombre del archivo no coincide exactamente con el nombre de la clase `public` en el archivo, o que escribió el nombre de la clase en forma incorrecta al momento de compilarla.

La figura 2.2 muestra el programa de la figura 2.1 ejecutándose en una ventana **Símbolo del sistema** de Microsoft® Windows® XP. Para ejecutar el programa, escriba `java Bienvenido1`; posteriormente se iniciará la JVM, que cargará el archivo "`.class`" para la clase `Bienvenido1`. Observe que la extensión "`.class`" del nombre de archivo se omite en el comando anterior; de no ser así, la JVM no ejecutaría el programa. La JVM llama al método `main`. A continuación, la instrucción de la línea 9 de `main` muestra `"Bienvenido a la programacion en Java!"` [*Nota:* muchos entornos muestran los símbolos del sistema con fondos negros y texto blanco. En nuestro entorno, ajustamos esta configuración para que nuestras capturas de pantalla fueran más legibles].

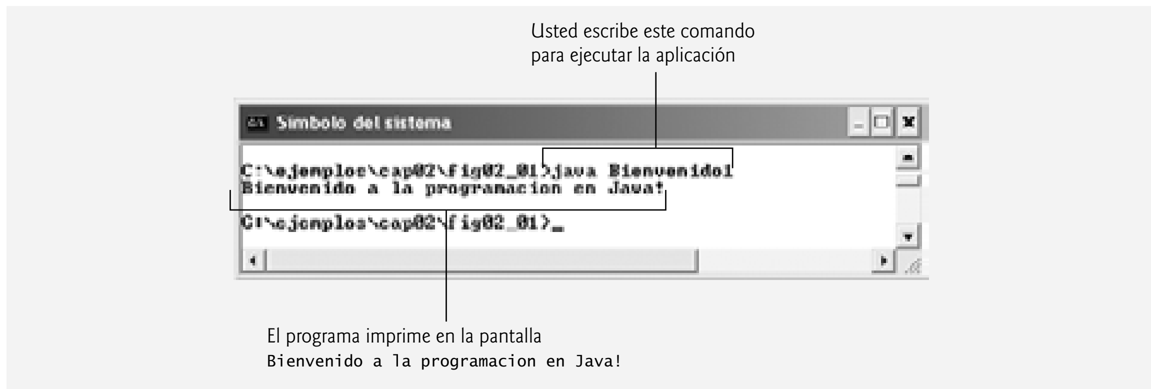


Figura 2.2 | Ejecución de `Bienvenido1` en una ventana **Símbolo del sistema** de Microsoft Windows XP.



Tip para prevenir errores 2.6

Al tratar de ejecutar un programa en Java, si recibe el mensaje "`Exception in thread "main" java.lang.NoClassDefFoundError: Bienvenido1`*", quiere decir que su variable de entorno* `CLASSPATH` *no se ha configurado apropiadamente. Consulte cuidadosamente las instrucciones de instalación en la sección Antes de empezar de este libro. En algunos sistemas, tal vez necesite reiniciar su equipo o abrir una nueva ventana de comandos para que estos ajustes tengan efecto.*

2.3 Modificación de nuestro primer programa en Java

Esta sección continúa con nuestra introducción a la programación en Java, con dos ejemplos que modifican el ejemplo de la figura 2.1 para imprimir texto en una línea utilizando varias instrucciones, y para imprimir texto en varias líneas utilizando una sola instrucción.

Cómo mostrar una sola línea de texto con varias instrucciones

`Bienvenido a la programacion en Java!` puede mostrarse en varias formas. La clase `Bienvenido2`, que se muestra en la figura 2.3, utiliza dos instrucciones para producir el mismo resultado que el de la figura 2.1. De aquí en adelante, resaltaremos las características nuevas y las características clave en cada listado de código, como se muestra en las línea 9 a 10 de este programa.

```
1  // Fig. 2.3: Bienvenido2.java
2  // Imprimir una línea de texto con varias instrucciones.
3
4  public class Bienvenido2
5  {
6     // el método main empieza la ejecución de la aplicación en Java
7     public static void main( String args[] )
8     {
9        System.out.print( "Bienvenido a " );
10       System.out.println( "la programacion en Java!" );
11
12    } // fin del método main
13
14 } // fin de la clase Bienvenido2
```

```
Bienvenido a la programacion en Java!
```

Figura 2.3 | Impresión de una línea de texto con varias instrucciones.

El programa es similar al de la figura 2.1, por lo que aquí sólo hablaremos de los cambios. La línea 2

```
// Imprimir una línea de texto con varias instrucciones.
```

es un comentario de fin de línea que describe el propósito de este programa. La línea 4 comienza la declaración de la clase `Bienvenido2`.

Las líneas 9 y 10 del método `main`

```
System.out.print( "Bienvenido a " );
System.out.println( "la programacion en Java!" );
```

muestran una línea de texto en la ventana de comandos. La primera instrucción utiliza el método `print` de `System.out` para mostrar una cadena. A diferencia de `println`, después de mostrar su argumento, `print` no posiciona el cursor de salida al inicio de la siguiente línea en la ventana de comandos; sino que el siguiente carácter aparecerá inmediatamente después del último que muestre `print`. Por lo tanto, la línea 10 coloca el primer carácter de su argumento (la letra “`l`”) inmediatamente después del último que muestra la línea 9 (el carácter de espacio antes del carácter de comilla doble de cierre de la cadena). Cada instrucción `print` o `println` continúa mostrando caracteres a partir de donde la última instrucción `print` o `println` dejó de mostrar caracteres.

Cómo mostrar varias líneas de texto con una sola instrucción

Una sola instrucción puede mostrar varias líneas, utilizando **caracteres de nueva línea**, los cuales indican a los métodos `print` y `println` de `System.out` cuándo deben colocar el cursor de salida al inicio de la siguiente línea en la ventana de comandos. Al igual que las líneas en blanco, los espacios y los tabuladores, los caracteres de nueva línea son caracteres de espacio en blanco. La figura 2.4 muestra cuatro líneas de texto, utilizando caracteres de nueva línea para determinar cuándo empezar cada nueva línea. La mayor parte del programa es idéntico a los de las figuras 2.1 y 2.3, por lo que aquí sólo veremos los cambios.

La línea 2

```
// Imprimir varias líneas de texto con una sola instrucción.
```

es un comentario que describe el propósito de este programa. La línea 4 comienza la declaración de la clase `Bienvenido3`.

La línea 9

```
System.out.println( "Bienvenido\na\nla programacion\nenJava!" );
```

muestra cuatro líneas separadas de texto en la ventana de comandos. Por lo general, los caracteres en una cadena se muestran exactamente como aparecen en las comillas dobles. Sin embargo, observe que los dos caracteres

```
1  // Fig. 2.4: Bienvenido3.java
2  // Imprimir varias líneas de texto con una sola instrucción.
3
4  public class Bienvenido3
5  {
6     // el método main empieza la ejecución de la aplicación en Java
7     public static void main( String args[] )
8     {
9        System.out.println( "Bienvenido\na\nla programacion\nen Java!" );
10
11    } // fin del método main
12
13 } // fin de la clase Bienvenido3
```

```
Bienvenido
a
la programacion
en Java!
```

Figura 2.4 | Impresión de varias líneas de texto con una sola instrucción.

Secuencia de escape	Descripción
\n	Nueva línea. Coloca el cursor de la pantalla al inicio de la siguiente línea.
\t	Tabulador horizontal. Desplaza el cursor de la pantalla hasta la siguiente posición de tabulación.
\r	Retorno de carro. Coloca el cursor de la pantalla al inicio de la línea actual; no avanza a la siguiente línea. Cualquier carácter que se imprima después del retorno de carro sobrescribe los caracteres previamente impresos en esa línea.
\\	Barra diagonal inversa. Se usa para imprimir un carácter de barra diagonal inversa.
\"	Doble comilla. Se usa para imprimir un carácter de doble comilla. Por ejemplo, `System.out.println( "\"entre comillas\"" );` muestra `"entre comillas"`

Figura 2.5 | Algunas secuencias de escape comunes.

\ y n (que se repiten tres veces en la instrucción) no aparecen en la pantalla. La **barra diagonal inversa** (\) se conoce como **carácter de escape**. Este carácter indica a los métodos `print` y `println` de `System.out` que se va a imprimir un "carácter especial". Cuando aparece una barra diagonal inversa en una cadena de caracteres, Java combina el siguiente carácter con la barra diagonal inversa para formar una **secuencia de escape**. La secuencia de escape \n representa el carácter de nueva línea. Cuando aparece un carácter de nueva línea en una cadena que se va a imprimir con `System.out`, el carácter de nueva línea hace que el cursor de salida de la pantalla se desplace al inicio de la siguiente línea en la ventana de comandos. En la figura 2.5 se enlistan varias secuencias de escape comunes, con descripciones de cómo afectan la manera de mostrar caracteres en la ventana de comandos. Para obtener una lista completa de secuencias de escape, visite `java.sun.com/docs/books/jls/third_edition/html/lexical.html#3.10.6`.

2.4 Cómo mostrar texto con printf

Java SE 5.0 agregó el método **`System.out.printf`** para mostrar datos con formato; la `f` en el nombre `printf` representa la palabra "formato". La figura 2.6 muestra las cadenas `"Bienvenido a"` y `"la programacion en Java!"` con `System.out.printf`.

Las líneas 9 y 10

```
System.out.printf( "%s\n%s\n",
   "Bienvenido a", "la programacion en Java!" );
```

llaman al método `System.out.printf` para mostrar la salida del programa. La llamada al método especifica tres argumentos. Cuando un método requiere varios argumentos, éstos se separan con comas (`,`); a esto se le conoce como **lista separada por comas**.



Buena práctica de programación 2.9

Coloque un espacio después de cada coma (,) en una lista de argumentos, para que sus programas sean más legibles.

Recuerde que todas las instrucciones en Java terminan con un punto y coma (`;`). Por lo tanto, las líneas 9 y 10 sólo representan una instrucción. Java permite que las instrucciones largas se dividan en varias líneas. Sin embargo, no puede dividir una instrucción a la mitad de un identificador, o de una cadena.



Error común de programación 2.7

Dividir una instrucción a la mitad de un identificador o de una cadena es un error de sintaxis.

```
1  // Fig. 2.6: Bienvenido4.java
2  // Imprimir varias líneas en un cuadro de diálogo.
3
4  public class Bienvenido4
5  {
6     // el método main empieza la ejecución de la aplicación de Java
7     public static void main( String args[] )
8     {
9        System.out.printf( "%s\n%s\n",
10          "Bienvenido a", "la programacion en Java!" );
11
12    } // fin del método main
13
14 } // fin de la clase Bienvenido4
```

```
Bienvenido a
la programacion en Java!
```

Figura 2.6 | Imprimir varias líneas de texto con el método `System.out.printf`.

El primer argumento del método `printf` es una **cadena de formato** que puede consistir en **texto fijo** y **especificadores de formato**. El método `printf` imprime el texto fijo de igual forma que `print` o `println`. Cada especificador de formato es un receptáculo para un valor, y especifica el tipo de datos a imprimir. Los especificadores de formato también pueden incluir información de formato opcional.

Los especificadores de formato empiezan con un signo porcentual (%) y van seguidos de un carácter que representa el tipo de datos. Por ejemplo, el especificador de formato **`%s`** es un receptáculo para una cadena. La cadena de formato en la línea 9 especifica que `printf` debe imprimir dos cadenas, y que a cada cadena le debe seguir un carácter de nueva línea. En la posición del primer especificador de formato, `printf` sustituye el valor del primer argumento después de la cadena de formato. En cada posición posterior de los especificadores de formato, `printf` sustituye el valor del siguiente argumento en la lista. Así, este ejemplo sustituye `"Bienvenido a"` por el primer `%s` y `"la programacion en Java!"` por el segundo `%s`. La salida muestra que se imprimieron dos líneas de texto.

En nuestros ejemplos, presentaremos las diversas características de formato a medida que se vayan necesitando. El capítulo 29 presenta los detalles de cómo dar formato a la salida con `printf`.

2.5 Otra aplicación en Java: suma de enteros

Nuestra siguiente aplicación lee (o recibe como entrada) dos **enteros** (números completos, como –22, 7, 0 y 1024) introducidos por el usuario mediante el teclado, calcula la suma de los valores y muestra el resultado. Este programa debe llevar la cuenta de los números que suministra el usuario para los cálculos que el programa realiza posteriormente. Los programas recuerdan números y otros datos en la memoria de la computadora, y acceden a esos datos a través de elementos del programa, conocidos como **variables**. El programa de la figura 2.7 demuestra estos conceptos. En los resultados de ejemplo, resaltamos las diferencias entre la entrada del usuario y la salida del programa.

```
// Fig. 2.7: Suma.java
// Programa que muestra la suma de dos enteros.
import java.util.Scanner; // el programa usa la clase Scanner

public class Suma
{
   // el método main empieza la ejecución de la aplicación en Java
```

Figura 2.7 | Programa que muestra la suma de dos enteros. (Parte 1 de 2).

```
8       public static void main( String args[] )
9       {
10         // crea objeto Scanner para obtener la entrada de la ventana de comandos
11         Scanner entrada = new Scanner( System.in );
12
13         int numero1; // primer número a sumar
14         int numero2; // segundo número a sumar
15         int suma; // suma de numero1 y numero2
16
17         System.out.print( "Escriba el primer entero: " ); // indicador
18         numero1 = entrada.nextInt(); // lee el primer número del usuario
19
20         System.out.print( "Escriba el segundo entero: " ); // indicador
21         numero2 = entrada.nextInt(); // lee el segundo número del usuario
22
23         suma = numero1 + numero2; // suma los números
24
25         System.out.printf( "La suma es %d\n", suma ); // muestra la suma
26
27      } // fin del método main
28
29   } // fin de la clase Suma
```

```
Escriba el primer entero: 45
Escriba el segundo entero: 72
La suma es 117
```

Figura 2.7 | Programa que muestra la suma de dos enteros. (Parte 2 de 2).

Las líneas 1 y 2

```
// Fig. 2.7: Suma.java
// Programa que muestra la suma de dos enteros.
```

indican el número de la figura, el nombre del archivo y el propósito del programa. La línea 3

```
import java.util.Scanner; // el programa usa la clase Scanner
```

es una **declaración `import`** que ayuda al compilador a localizar una clase que se utiliza en este programa. Una gran fortaleza de Java es su extenso conjunto de clases predefinidas que podemos utilizar, en vez de "reinventar la rueda". Estas clases se agrupan en **paquetes** (colecciones con nombre de clases relacionadas) y se conocen en conjunto como la **biblioteca de clases de Java**, o **Interfaz de Programación de Aplicaciones de Java** (**API de Java**). Los programadores utilizan declaraciones `import` para identificar las clases predefinidas que se utilizan en un programa de Java. La declaración `import` en la línea 3 indica que este ejemplo utiliza la clase `Scanner` predefinida de Java (que veremos en breve) del paquete **`java.util`**. Después, el compilador trata de asegurarse que utilicemos la clase `Scanner` de manera apropiada.



Error común de programación 2.8

Todas las declaraciones `import` deben aparecer antes de la primera declaración de clase en el archivo. Colocar una declaración `import` dentro del cuerpo de la declaración de una clase, o después de la declaración de la misma, es un error de sintaxis.



Tip para prevenir errores 2.7

Por lo general, si olvida incluir una declaración `import` para una clase que utilice en su programa, se produce un error de compilación que contiene el mensaje: "`cannot resolve symbol`". Cuando esto ocurra, verifique que haya proporcionado las declaraciones `import` apropiadas y que los nombres en las mismas estén escritos correctamente, incluyendo el uso apropiado de las letras mayúsculas y minúsculas.

La línea 5

```
public class Suma
```

empieza la declaración de la clase `Suma`. El nombre de archivo para esta clase `public` debe ser `Suma.java`. Recuerde que el cuerpo de cada declaración de clase empieza con una llave izquierda de apertura (línea 6) (`{`) y termina con una llave derecha de cierre (línea 29) (`}`).

La aplicación empieza a ejecutarse con el método `main` (líneas 8 a la 27). La llave izquierda (línea 9) marca el inicio del cuerpo de `main`, y la correspondiente llave derecha (línea 27) marca el final de `main`. Observe que al método `main` se le aplica un nivel de sangría en el cuerpo de la clase `Suma`, y que al código en el cuerpo de `main` se le aplica otro nivel para mejorar la legibilidad.

La línea 11

```
Scanner entrada = new Scanner( System.in );
```

es una **instrucción de declaración de variable** (también conocida como **declaración**), la cual especifica el nombre (`entrada`) y tipo (`Scanner`) de una variable utilizada en este programa. Una **variable** es una ubicación en la memoria de la computadora, en donde se puede guardar un valor para utilizarlo posteriormente en un programa. Todas las variables deben declararse con un **nombre** y un **tipo** antes de poder usarse; este nombre permite al programa acceder al valor de la variable en memoria; y puede ser cualquier identificador válido. (Consulte en la sección 2.2 los requerimientos para nombrar identificadores). El tipo de una variable especifica el tipo de información que se guarda en esa ubicación de memoria. Al igual que las demás instrucciones, las instrucciones de declaración terminan con punto y coma (`;`).

La declaración en la línea 11 especifica que la variable llamada `entrada` es de tipo `Scanner`. Un objeto **`Scanner`** permite a un programa leer datos (como números) para usarlos. Los datos pueden provenir de muchas fuentes, como un archivo en disco, o desde el teclado. Antes de usar un objeto `Scanner`, el programa debe crearlo y especificar el origen de los datos.

El signo igual (=) en la línea 11 indica que la variable `entrada` tipo `Scanner` debe **inicializarse** (es decir, hay que prepararla para usarla en el programa) en su declaración con el resultado de la expresión **`new Scanner ( System.in )`** a la derecha del signo igual. Esta expresión crea un objeto `Scanner` que lee los datos escritos por el usuario mediante el teclado. Recuerde que el objeto de salida estándar, `System.out`, permite a las aplicaciones de Java mostrar caracteres en la ventana de comandos. De manera similar, el **objeto de entrada estándar**, `System.in`, permite a las aplicaciones de Java leer la información escrita por el usuario. Así, la línea 11 crea un objeto `Scanner` que permite a la aplicación leer la información escrita por el usuario mediante el teclado.

Las instrucciones de declaración de variables en las líneas 13 a la 15

```
int numero1; // primer número a sumar
int numero2; // segundo número a sumar
int suma; // suma de numero1 y numero2
```

declaran que las variables `numero1`, `numero2` y `suma` contienen datos de tipo **`int`**; estas variables pueden contener valores **enteros** (números completos, como `7`, `–11`, `0` y `31,914`). Estas variables no se han inicializado todavía. El rango de valores para un `int` es de –2,147,483,648 a +2,147,483,647. Pronto hablaremos sobre los tipos **`float`** y **`double`**, para guardar números reales, y sobre el tipo **`char`**, para guardar datos de caracteres. Los números reales son números que contienen puntos decimales, como `3.4`, `0.0` y `–11.19`. Las variables de tipo `char` representan caracteres individuales, como una letra en mayúscula (como `A`), un dígito (como `7`), un carácter especial (como `*` o `%`) o una secuencia de escape (como el carácter de nueva línea, `\n`). Los tipos como `int`, `float`, `double` y `char` se conocen como **tipos primitivos** o **tipos integrados**. Los nombres de los tipos primitivos son palabras clave y, por lo tanto, deben aparecer completamente en minúsculas. El apéndice D, Tipos primitivos, sintetiza las características de los ocho tipos primitivos (`boolean`, `byte`, `char`, `short`, `int`, `long`, `float` y `double`).

Las instrucciones de declaración de variables pueden dividirse en varias líneas, separando los nombres de las variables por comas (es decir, una lista de nombres de variables separados por comas). Varias variables del mismo tipo pueden declararse en una, o en varias declaraciones. Por ejemplo, las líneas 13 a la 15 se pueden escribir como una sola instrucción, de la siguiente manera:

```
int numero1, // primer número a sumar
    numero2, // segundo número a sumar
    suma; // suma de numero1 y numero2
```

Observe que utilizamos comentarios de fin de línea en las líneas 13 a la 15. Este uso de comentarios es una práctica común de programación, para indicar el propósito de cada variable en el programa.



Buena práctica de programación 2.10

Declare cada variable en una línea separada. Este formato permite insertar fácilmente un comentario descriptivo a continuación de cada declaración.



Buena práctica de programación 2.11

Seleccionar nombres de variables significativos ayuda a que un programa se **autodocumente** *(es decir, que sea más fácil entender con sólo leerlo, en lugar de leer manuales o ver un número excesivo de comentarios).*



Buena práctica de programación 2.12

Por convención, los identificadores de nombre de variables empiezan con una letra minúscula, y cada una de las palabras en el nombre, que van después de la primera, deben empezar con una letra mayúscula. Por ejemplo, el identificador `primerNumero` *tiene una* `N` *mayúscula en su segunda palabra,* `Numero`*.*

La línea 17

```
System.out.print( "Escriba el primer entero: " ); // indicador
```

utiliza `System.out.print` para mostrar el mensaje `"Escriba el primer entero: "`. Este mensaje se conoce como **indicador**, ya que indica al usuario que debe realizar una acción específica. En la sección 2.2 vimos que los identificadores que empiezan con letras mayúsculas representan nombres de clases. Por lo tanto, `System` es una clase; que forma parte del paquete **`java.lang`**. Observe que la clase `System` no se importa con una declaración `import` al principio del programa.



Observación de ingeniería de software 2.1

El paquete `java.lang` *se importa de manera predeterminada en todos los programas de Java; por ende, las clases en* `java.lang` *son las únicas en la API que no requieren una declaración* `import`*.*

La línea 18

```
numero1 = entrada.nextInt(); // lee el primer número del usuario
```

utiliza el método `nextInt` del objeto `entrada` de la clase `Scanner` para obtener un entero del usuario mediante el teclado. En este punto, el programa espera a que el usuario escriba el número y oprima *Intro* para enviar el número al programa.

Técnicamente, el usuario puede escribir cualquier cosa como valor de entrada. Nuestro programa asume que el usuario escribirá un valor de entero válido, según las indicaciones; si el usuario escribe un valor no entero, se producirá un error lógico en tiempo de ejecución y el programa no funcionará correctamente. El capítulo 13, Manejo de excepciones, habla sobre cómo hacer sus programas más robustos, al permitirles manejar dichos errores. Esto también se conoce como hacer que su programa sea **tolerante a fallas**.

En la línea 18, el resultado de la llamada al método `nextInt` (un valor `int`) se coloca en la variable `numero1` mediante el uso del **operador de asignación**, =. La instrucción se lee como "`numero1` obtiene el valor de `entrada.nextInt()`". Al operador `=` se le llama **operador binario**, ya que tiene dos **operandos**: `numero1` y el resultado de la llamada al método `entrada.nextInt()`. Esta instrucción se llama **instrucción de asignación**, ya que asigna un valor a una variable. Todo lo que está a la derecha del operador de asignación (=) se evalúa siempre antes de realizar la asignación.



Buena práctica de programación 2.13

Coloque espacios en cualquier lado de un operador binario, para que resalte y el programa sea más legible.

La línea 20

```
System.out.print( "Escriba el segundo entero: " ); // indicador
```

pide al usuario que escriba el segundo entero.

La línea 21

```
numero2 = entrada.nextInt(); // lee el segundo número del usuario
```

lee el segundo entero y lo asigna a la variable `numero2`.

La línea 23

```
suma = numero1 + numero2; // suma los números
```

es una instrucción de asignación que calcula la suma de las variables `numero1` y `numero2`, y asigna el resultado a la variable `suma` mediante el uso del operador de asignación, =. La instrucción se lee como "`suma` obtiene el valor de `numero1 + numero2`". La mayoría de los cálculos se realizan en instrucciones de asignación. Cuando el programa encuentra la operación de suma, utiliza los valores almacenados en las variables `numero1` y `numero2` para realizar el cálculo. En la instrucción anterior, el operador de suma es binario; sus dos operandos son `numero1` y `numero2`. Las partes de las instrucciones que contienen cálculos se llaman **expresiones**. De hecho, una expresión es cualquier parte de una instrucción que tiene un valor asociado. Por ejemplo, el valor de la expresión `numero1 + numero2` es la suma de los números. De manera similar, el valor de la expresión `entrada.nextInt()` es un entero escrito por el usuario.

Una vez realizado el cálculo, la línea 25

```
System.out.printf( "La suma es %d\n", suma ); // muestra la suma
```

utiliza el método `System.out.printf` para mostrar la `suma`. El especificador de formato `%d` es un receptáculo para un valor `int` (en este caso, el valor de `suma`); la letra `d` representa "entero decimal". Observe que aparte del especificador de formato `%d`, el resto de los caracteres en la cadena de formato son texto fijo. Por lo tanto, el método `printf` imprime en pantalla `"La suma es "`, seguido del valor de `suma` (en la posición del especificador de formato `%d`) y una nueva línea.

Observe que los cálculos también pueden realizarse dentro de instrucciones `printf`. Podríamos haber combinado las instrucciones de las líneas 23 y 25 en la siguiente instrucción:

```
System.out.printf( "La suma es %d\n", ( numero1 + numero2 ) );
```

Los paréntesis alrededor de la expresión `numero1 + numero2` no son requeridos; se incluyen para enfatizar que el valor de la expresión se imprime en la posición del especificador de formato `%d`.

Documentación de la API de Java

Para cada nueva clase de la API de Java que utilizamos, indicamos el paquete en el que se ubica. Esta información es importante, ya que nos ayuda a localizar las descripciones de cada paquete y clase en la **documentación de la API de Java**. Puede encontrar una versión basada en Web de esta documentación en

```
java.sun.com/javase/6/docs/api/
```

También puede descargar esta documentación, en su propia computadora, de

```
java.sun.com/javase/downloads/ea.jsp
```

La descarga es de aproximadamente 53 megabytes (MB). El apéndice J, Uso de la documentación de la API de Java, describe cómo utilizar esta documentación.

2.6 Conceptos acerca de la memoria

Los nombres de variables como `numero1`, `numero2` y `suma` en realidad corresponden a ciertas **ubicaciones** en la memoria de la computadora. Toda variable tiene un **nombre**, un **tipo**, un **tamaño** y un **valor**.

En el programa de suma de la figura 2.7, cuando se ejecuta la instrucción (línea 18)

```
numero1 = entrada.nextInt(); // lee el primer número del usuario
```

el número escrito por el usuario se coloca en una ubicación de memoria a la cual se asigna el nombre `numero1`. Suponga que el usuario escribe `45`. La computadora coloca ese valor entero en la ubicación `numero1`, como se

numero1	45

Figura 2.8 | Ubicación de memoria que muestra el nombre y el valor de la variable `numero1`.

numero1	45
numero2	72

Figura 2.9 | Ubicaciones de memoria, después de almacenar valores para `numero1` y `numero2`.

numero1	45
numero2	72
suma	117

Figura 2.10 | Ubicaciones de memoria, después de almacenar la suma de `numero1` y `numero2`.

muestra en la figura 2.8. Cada vez que se coloca un nuevo valor en una ubicación de memoria, se sustituye al valor anterior en esa ubicación; es decir, el valor anterior se pierde.

Cuando se ejecuta la instrucción (línea 21)

```
numero2 = entrada.nextInt(); // lee el segundo número del usuario
```

suponga que el usuario escribe `72`. La computadora coloca ese valor entero en la ubicación `numero2`. La memoria ahora aparece como se muestra en la figura 2.9.

Una vez que el programa de la figura 2.7 obtiene valores para `numero1` y `numero2`, los suma y coloca el resultado en la variable `suma`. La instrucción (línea 23)

```
suma = numero1 + numero2; // suma los números
```

realiza la suma y después sustituye el valor anterior de `suma`. Una vez que se calcula `suma`, la memoria aparece como se muestra en la figura 2.10. Observe que los valores de `numero1` y `numero2` aparecen exactamente como antes de usarlos en el cálculo de `suma`. Estos valores se utilizaron, pero no se destruyeron, cuando la computadora realizó el cálculo. Por ende, cuando se lee un valor de una ubicación de memoria, el proceso es no destructivo.

2.7 Aritmética

La mayoría de los programas realizan cálculos aritméticos. Los **operadores aritméticos** se sintetizan en la figura 2.11. Observe el uso de varios símbolos especiales que no se utilizan en álgebra. El **asterisco** (*) indica la multiplicación, y el **signo de porcentaje** (%) es el **operador residuo** (**conocido** como **módulo** en algunos lenguajes), el cual describiremos en breve. Los operadores aritméticos en la figura 2.11 son binarios, ya que funcionan con dos operandos. Por ejemplo, la expresión `f + 7` contiene el operador binario `+` y los dos operandos `f` y `7`.

La **división de enteros** produce un cociente entero: por ejemplo, la expresión `7 / 4` da como resultado `1`, y la expresión `17 / 5` da como resultado `3`. Cualquier parte fraccionaria en una división de enteros simplemente se descarta (es decir, se trunca); no ocurre un redondeo. Java proporciona el operador residuo, `%`, el cual produce el residuo después de la división. La expresión `x % y` produce el residuo después de que `x` se divide entre `y`. Por lo tanto, `7 % 4` produce `3`, y `17 % 5` produce `2`. Por lo general, este operador se utiliza más con operandos enteros, pero también puede usarse con otros tipos aritméticos. En los ejercicios de este capítulo y de capítulos posteriores, consideramos muchas aplicaciones interesantes del operador residuo, como determinar si un número es múltiplo de otro.

Operación en Java	Operador aritmético	Expresión algebraica	Expresión en Java
Suma	+	$f + 7$	`f + 7`
Resta	-	$p - c$	`p - c`
Multiplicación	*	bm	`b * m`
División	/	x / y o $\frac{x}{y}$ o $x \div y$	`x / y`
Residuo	%	$r \bmod s$	`r % s`

Figura 2.11 | Operadores aritméticos.

Expresiones aritméticas en formato de línea recta

Las expresiones aritméticas en Java deben escribirse en **formato de línea recta** para facilitar la escritura de programas en la computadora. Por lo tanto, las expresiones como "`a` dividida entre `b`" deben escribirse como `a / b`, de manera que todas las constantes, variables y operadores aparezcan en una línea recta. La siguiente notación algebraica no es generalmente aceptable para los compiladores:

$$\frac{a}{b}$$

Paréntesis para agrupar subexpresiones

Los paréntesis se utilizan para agrupar términos en las expresiones en Java, de la misma manera que en las expresiones algebraicas. Por ejemplo, para multiplicar `a` por la cantidad `b + c`, escribimos

```
a * ( b + c )
```

Si una expresión contiene **paréntesis anidados**, como

```
( ( a + b ) * c )
```

se evalúa primero la expresión en el conjunto más interno de paréntesis (`a + b` en este caso).

Reglas de precedencia de operadores

Java aplica los operadores en expresiones aritméticas en una secuencia precisa, determinada por las siguientes **reglas de precedencia de operadores**, que generalmente son las mismas que las que se utilizan en álgebra (figura 2.12):

1. Las operaciones de multiplicación, división y residuo se aplican primero. Si una expresión contiene varias de esas operaciones, los operadores se aplican de izquierda a derecha. Los operadores de multiplicación, división y residuo tienen el mismo nivel de precedencia.
2. Las operaciones de suma y resta se aplican a continuación. Si una expresión contiene varias de esas operaciones, los operadores se aplican de izquierda a derecha. Los operadores de suma y resta tienen el mismo nivel de precedencia.

Operador(es)	Operación(es)	Orden de evaluación (precedencia)
* / %	Multiplicación División Residuo	Se evalúan primero. Si hay varios operadores de este tipo, se evalúan de izquierda a derecha.
+ -	Suma Resta	Se evalúan después. Si hay varios operadores de este tipo, se evalúan de izquierda a derecha.

Figura 2.12 | Precedencia de los operadores aritméticos.

Estas reglas permiten a Java aplicar los operadores en el orden correcto. Cuando decimos que los operadores se aplican de izquierda a derecha, nos referimos a su **asociatividad**; veremos que algunos se asocian de derecha a izquierda. La figura 2.12 sintetiza estas reglas de precedencia de operadores; esta tabla se expandirá a medida que se introduzcan más operadores en Java. En el apéndice A, Tabla de precedencia de los operadores, se incluye una tabla de precedencias completa.

Ejemplos de expresiones algebraicas y de Java

Ahora, consideremos varias expresiones en vista de las reglas de precedencia de operadores. Cada ejemplo enlista una expresión algebraica y su equivalente en Java. El siguiente es un ejemplo de una media (promedio) aritmética de cinco términos:

Álgebra: $m = \frac{a + b + c + d + e}{5}$

Java: `m = ( a + b + c + d + e ) / 5;`

Los paréntesis son obligatorios, ya que la división tiene una mayor precedencia que la suma. La cantidad completa `( a + b + c + d + e )` va a dividirse entre 5. Si por error se omiten los paréntesis, obtenemos `a + b + c + d + e / 5`, lo cual da como resultado

$$a + b + c + d + \frac{e}{5}$$

El siguiente es un ejemplo de una ecuación de línea recta:

Álgebra: $y = mx + b$

Java: `y = m * x + b;`

No se requieren paréntesis. El operador de multiplicación se aplica primero, ya que la multiplicación tiene mayor precedencia sobre la suma. La asignación ocurre al último, ya que tiene menor precedencia que la multiplicación o suma.

El siguiente ejemplo contiene las operaciones residuo (%), multiplicación, división, suma y resta:

Álgebra: $z = pr\%q + w/x - y$

Java: `z = p * r % q + w / x - y;`



Los números dentro de los círculos bajo la instrucción, indican el orden en el que Java aplica los operadores. Las operaciones de multiplicación, residuo y división se evalúan primero, en orden de izquierda a derecha (es decir, se asocian de izquierda a derecha), ya que tienen mayor precedencia que la suma y la resta. Las operaciones de suma y resta se evalúan a continuación; estas operaciones también se aplican de izquierda a derecha.

Evaluación de un polinomio de segundo grado

Para desarrollar una mejor comprensión de las reglas de precedencia de operadores, considere la evaluación de un polinomio de segundo grado ($y = ax^2 + bx + c$):

```
y = a * x * x + b * x + c;
```



Los números dentro de los círculos indican el orden en el que Java aplica los operadores. Las operaciones de multiplicación se evalúan primero en orden de izquierda a derecha (es decir, se asocian de izquierda a derecha), ya que tienen mayor precedencia que la suma. Las operaciones de suma se evalúan a continuación y se aplican de izquierda a derecha. No existe un operador aritmético para la potencia de un número en Java, por lo que x^2 se representa como `x * x`. La sección 5.4 muestra una alternativa para calcular la potencia de un número en Java.

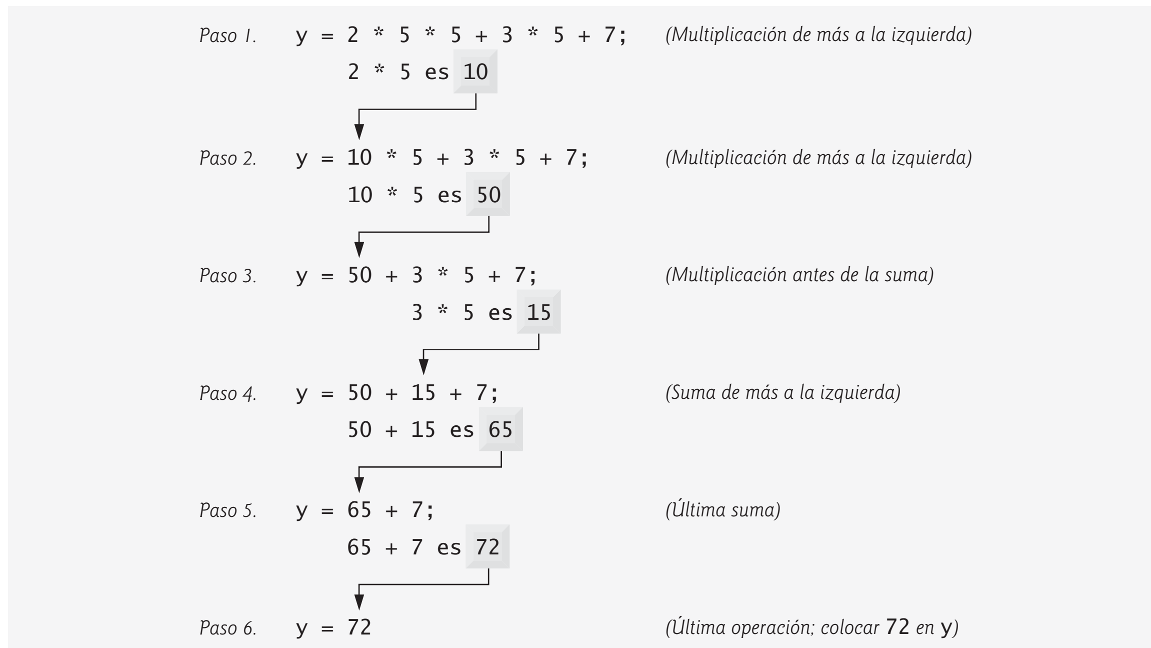


Figura 2.13 | Orden en el cual se evalúa un polinomio de segundo grado.

Suponga que `a, b, c` y `x` en el polinomio de segundo grado anterior se inicializan (reciben valores) como sigue: `a = 2, b = 3, c = 7` y `x = 5`. La figura 2.13 muestra el orden en el que se aplican los operadores.

Al igual que en álgebra, es aceptable colocar paréntesis innecesarios en una expresión para hacer que ésta sea más clara. A dichos paréntesis se les llama **paréntesis redundantes**. Por ejemplo, la instrucción de asignación anterior podría colocarse entre paréntesis, de la siguiente manera:

```
y = ( a * x * x ) + ( b * x ) + c;
```



Buena práctica de programación 2.14

El uso de paréntesis para las expresiones aritméticas complejas, incluso cuando éstos no sean necesarios, puede hacer que las expresiones aritméticas sean más fáciles de leer.

2.8 Toma de decisiones: operadores de igualdad y relacionales

Una **condición** es una expresión que puede ser **verdadera** (**`true`**) o **falsa** (**`false`**). Esta sección presenta la **instrucción `if`** de Java, la cual permite que un programa tome una **decisión**, con base en el valor de una condición. Por ejemplo, la condición "calificación es mayor o igual que 60" determina si un estudiante pasó o no una prueba. Si la condición en una instrucción `if` es verdadera, el cuerpo de la instrucción `if` se ejecuta. Si la condición es falsa, el cuerpo no se ejecuta. Veremos un ejemplo en breve.

Las condiciones en las instrucciones `if` pueden formarse utilizando los **operadores de igualdad** (`==` y `!=`) y los **operadores relacionales** (`>`, `<`, `>=` y `<=`) que se sintetizan en la figura 2.14. Ambos operadores de igualdad tienen el mismo nivel de precedencia, que es menor que la precedencia de los operadores relacionales. Los operadores de igualdad se asocian de izquierda a derecha; y los relacionales tienen el mismo nivel de precedencia y también se asocian de izquierda a derecha.

La aplicación de la figura 2.15 utiliza seis instrucciones `if` para comparar dos enteros introducidos por el usuario. Si la condición en cualquiera de estas instrucciones `if` es verdadera, se ejecuta la instrucción de asignación asociada. El programa utiliza un objeto `Scanner` para recibir los dos enteros del usuario y almacenarlos en las variables `numero1` y `numero2`. Después, compara los números y muestra los resultados de las comparaciones que son verdaderas.

Operador estándar algebraico de igualdad o relacional	Operador de igualdad o relacional de Java	Ejemplo de condición en Java	Significado de la condición en Java
Operadores de igualdad			
=	==	x == y	x es igual a y
≠	!=	x != y	x no es igual a y
Operadores relacionales			
>	>	x > y	x es mayor que y
<	<	x < y	x es menor que y
≥	>=	x >= y	x es mayor o igual que y
≤	<=	x <= y	x es menor o igual que y

Figura 2.14 | Operadores de igualdad y relacionales.

```
// Fig. 2.15: Comparacion.java
// Compara enteros utilizando instrucciones if, operadores relacionales
// y de igualdad.
import java.util.Scanner; // el programa utiliza la clase Scanner

public class Comparacion
{
   // el método main empieza la ejecución de la aplicación en Java
   public static void main( String args[] )
   {
      // crea objeto Scanner para obtener la entrada de la ventana de comandos
      Scanner entrada = new Scanner( System.in );

      int numero1; // primer número a comparar
      int numero2; // segundo número a comparar

      System.out.print( "Escriba el primer entero: " ); // indicador
      numero1 = entrada.nextInt(); // lee el primer número del usuario

      System.out.print( “Escriba el segundo entero: “ ); // indicador
      numero2 = entrada.nextInt(); // lee el segundo número del usuario

      if ( numero1 == numero2 )
         System.out.printf( “%d == %d\n”, numero1, numero2 );

      if ( numero1 != numero2 )
         System.out.printf( “%d != %d\n”, numero1, numero2 );

      if ( numero1 < numero2 )
         System.out.printf( “%d < %d\n”, numero1, numero2 );

      if ( numero1 > numero2 )
         System.out.printf( “%d > %d\n”, numero1, numero2 );

      if ( numero1 <= numero2 )
         System.out.printf( “%d <= %d\n”, numero1, numero2 );
```

Figura 2.15 | Operadores de igualdad y relacionales. (Parte 1 de 2).

```
37
38          if ( numero1 >= numero2 )
39             System.out.printf( “%d >= %d\n”, numero1, numero2 );
40
41      } // fin del método main
42
43   } // fin de la clase Comparacion
```

```
Escriba el primer entero: 777
Escriba el segundo entero: 777
777 == 777
777 <= 777
777 >= 777
```

```
Escriba el primer entero: 1000
Escriba el segundo entero: 2000
1000 != 2000
1000 < 2000
1000 <= 2000
```

```
Escriba el primer entero: 2000
Escriba el segundo entero: 1000
2000 != 1000
2000 > 1000
2000 >= 1000
```

Figura 2.15 | Operadores de igualdad y relacionales. (Parte 2 de 2).

La declaración de la clase `Comparacion` comienza en la línea 6

```
public class Comparacion
```

El método `main` de la clase (líneas 9 a 41) empieza la ejecución del programa. La línea 12

```
Scanner entrada = new Scanner( System.in );
```

declara la variable `entrada` de la clase `Scanner` y le asigna un objeto `Scanner` que recibe datos de la entrada estándar (es decir, el teclado).

Las líneas 14 y 15

```
int numero1; // primer número a comparar
int numero2; // segundo número a comparar
```

declaran las variables `int` que se utilizan para almacenar los valores introducidos por el usuario.

Las líneas 17-18

```
System.out.print( "Escriba el primer entero: " ); // indicador
numero1 = entrada.nextInt(); // lee el primer número del usuario
```

piden al usuario que introduzca el primer entero y el valor, respectivamente. El valor de entrada se almacena en la variable `numero1`.

Las líneas 20-21

```
System.out.print( "Escriba el segundo entero: " ); // indicador
numero2 = entrada.nextInt(); // lee el segundo número del usuario
```

piden al usuario que introduzca el segundo entero y el valor, respectivamente. El valor de entrada se almacena en la variable `numero2`.

Las líneas 23-24

```
if ( numero1 == numero2 )
    System.out.printf( "%d == %d\n", numero1, numero2 );
```

declaran una instrucción `if` que compara los valores de las variables `numero1` y `numero2`, para determinar si son iguales o no. Una instrucción `if` siempre empieza con la palabra clave `if`, seguida de una condición entre paréntesis. Una instrucción `if` espera una instrucción en su cuerpo. La sangría de la instrucción del cuerpo que se muestra aquí no es obligatoria, pero mejora la legibilidad del programa al enfatizar que la instrucción en la línea 24 forma parte de la instrucción `if` que empieza en la línea 23. La línea 24 sólo se ejecuta si los números almacenados en las variables `numero1` y `numero2` son iguales (es decir, si la condición es verdadera). Las instrucciones `if` en las líneas 26-27, 29-30, 32-33, 35-36 y 38-39 comparan a `numero1` y `numero2` con los operadores `!=`, `<`, `>`, `<=` y `>=`, respectivamente. Si la condición en cualquiera de las instrucciones `if` es verdadera, se ejecuta la instrucción del cuerpo correspondiente.



Error común de programación 2.9

Olvidar los paréntesis izquierdo y/o derecho de la condición en una instrucción `if` *es un error de sintaxis; los paréntesis son obligatorios.*



Error común de programación 2.10

Confundir el operador de igualdad (==) con el de asignación (=) puede producir un error lógico o de sintaxis. El operador de igualdad debe leerse como "es igual a", y el de asignación como "obtiene" u "obtiene el valor de". Para evitar confusión, algunas personas leen el operador de igualdad como "doble igual" o "igual igual".



Error común de programación 2.11

Es un error de sintaxis si los operadores ==, !=, >= y <= contienen espacios entre sus símbolos, como en = =, ! =, > = y < =, respectivamente.



Error común de programación 2.12

Invertir los operadores !=, >= y <=, como en =!, => y =<, es un error de sintaxis.



Buena práctica de programación 2.15

Aplique sangría al cuerpo de una instrucción `if` *para hacer que resalte y mejorar la legibilidad del programa.*



Buena práctica de programación 2.16

Coloque sólo una instrucción por línea en un programa. Este formato mejora la legibilidad del programa.

Observe que no hay punto y coma (`;`) al final de la primera línea de cada instrucción `if`. Dicho punto y coma produciría un error lógico en tiempo de compilación. Por ejemplo,

```
if ( numero1 == numero2 );  // error lógico
    System.out.printf( "%d == %d\n", numero1, numero2 );
```

sería interpretada por Java de la siguiente manera:

```
if ( numero1 == numero2 )
    ; // instrucción vacía

System.out.printf( "%d == %d\n", numero1, numero2 );
```

en donde el punto y coma que aparece por sí solo en una línea (que se conoce como **instrucción vacía** o **nula**) es la instrucción que se va a ejecutar si la condición en la instrucción `if` es verdadera. Al ejecutarse la instrucción vacía, no se lleva a cabo ninguna tarea en el programa. Éste continúa con la instrucción de salida, que siempre se

Operadores	Asociatividad	Tipo
`*` `/` `%`	izquierda a derecha	multiplicativa
`+` `-`	izquierda a derecha	suma
`<` `<=` `>` `>=`	izquierda a derecha	relacional
`==` `!=`	izquierda a derecha	igualdad
`=`	derecha a izquierda	asignación

Figura 2.16 | Precedencia y asociatividad de los operadores descritos hasta ahora.

ejecuta, sin importar que la condición sea verdadera o falsa, ya que la instrucción de salida no forma parte de la instrucción `if`.



Error común de programación 2.13

Colocar un punto y coma inmediatamente después del paréntesis derecho de la condición en una instrucción `if` *es, generalmente, un error lógico.*

Observe el uso del espacio en blanco en la figura 2.15. Recuerde que los caracteres de espacio en blanco, como tabuladores, nuevas líneas y espacios, generalmente son ignorados por el compilador. Por lo tanto, las instrucciones pueden dividirse en varias líneas y pueden espaciarse de acuerdo a las preferencias del programador, sin afectar el significado de un programa. Es incorrecto dividir identificadores y cadenas. Idealmente las instrucciones deben mantenerse lo más reducidas que sea posible, pero no siempre se puede hacer esto.



Buena práctica de programación 2.17

Una instrucción larga puede esparcirse en varias líneas. Si una sola instrucción debe dividirse entre varias líneas, los puntos que elija para hacer la división deben tener sentido, como después de una coma en una lista separada por comas, o después de un operador en una expresión larga. Si una instrucción se divide entre dos o más líneas, aplique sangría a todas las líneas subsecuentes hasta el final de la instrucción.

La figura 2.16 muestra la precedencia de los operadores que se presentan en este capítulo. Los operadores se muestran de arriba hacia abajo, en orden descendente de precedencia; todos, con la excepción del operador de asignación, `=`, se asocian de izquierda a derecha. La suma es asociativa a la izquierda, por lo que una expresión como `x + y + z` se evalúa como si se hubiera escrito así: `( x + y ) + z`. El operador de asignación, `=`, asocia de derecha a izquierda, por lo que una expresión como `x = y = 0` se evalúa como si se hubiera escrito así: `x = ( y = 0 )`, en donde, como pronto veremos, primero se asigna el valor `0` a la variable `y`, y después se asigna el resultado de esa asignación, `0`, a `x`.



Buena práctica de programación 2.18

Cuando escriba expresiones que contengan muchos operadores, consulte la tabla de precedencia (apéndice A). Confirme que las operaciones en la expresión se realicen en el orden que usted espera. Si no está seguro acerca del orden de evaluación en una expresión compleja, utilice paréntesis para forzar el orden, en la misma forma que lo haría con las expresiones algebraicas. Observe que algunos operadores como el de asignación, `=`*, asocian de derecha a izquierda, en vez de hacerlo de izquierda a derecha.*

2.9 (Opcional) Ejemplo práctico de Ingeniería de Software: cómo examinar el documento de requerimientos de un problema

Ahora empezaremos nuestro ejemplo práctico opcional de diseño e implementación orientados a objetos. Las secciones del Ejemplo práctico de Ingeniería de Software al final de este y los siguientes capítulos le ayudarán a incursionar en la orientación a objetos, mediante el análisis de un ejemplo práctico de una máquina de cajero automático

(Automated Teller Machine o ATM, por sus siglas en inglés). Este ejemplo práctico le brindará una experiencia de diseño e implementación substancial, cuidadosamente pautada y completa. En los capítulos 3 al 8 y 10, llevaremos a cabo los diversos pasos de un proceso de diseño orientado a objetos (DOO) utilizando UML, mientras relacionamos estos pasos con los conceptos orientados a objetos que se describen en los capítulos. El apéndice M implementa el ATM utilizando las técnicas de la programación orientada a objetos (POO) en Java. Presentaremos la solución completa al ejemplo práctico. Éste no es un ejercicio, sino una experiencia de aprendizaje de extremo a extremo, que concluye con un análisis detallado del código en Java que implementamos, con base en nuestro diseño. Este ejemplo práctico le ayudará a acostumbrarse a los tipos de problemas substanciales que se encuentran en la industria, y sus soluciones potenciales. Esperamos que disfrute esta experiencia de aprendizaje.

Empezaremos nuestro proceso de diseño con la presentación de un **documento de requerimientos**, el cual especifica el propósito general del sistema ATM y *qué* debe hacer. A lo largo del ejemplo práctico, nos referiremos al documento de requerimientos para determinar con precisión la funcionalidad que debe incluir el sistema.

Documento de requerimientos

Un banco local pretende instalar una nueva máquina de cajero automático (ATM), para permitir a los usuarios (es decir, los clientes del banco) realizar transacciones financieras básicas (figura 2.17). Cada usuario sólo puede tener una cuenta en el banco. Los usuarios del ATM deben poder ver el saldo de su cuenta, retirar efectivo (es decir, sacar dinero de una cuenta) y depositar fondos (es decir, meter dinero en una cuenta). La interfaz de usuario del cajero automático contiene los siguientes componentes:

- una pantalla que muestra mensajes al usuario
- un teclado que recibe datos numéricos de entrada del usuario
- un dispensador de efectivo que dispensa efectivo al usuario, y
- una ranura de depósito que recibe sobres para depósitos del usuario.

El dispensador de efectivo comienza cada día cargado con 500 billetes de $20. [*Nota*: debido al alcance limitado de este ejemplo práctico, ciertos elementos del ATM que se describen aquí no imitan exactamente a los de un ATM real. Por ejemplo, generalmente un ATM contiene un dispositivo que lee el número de cuenta del usuario de una tarjeta para ATM, mientras que este ATM pide al usuario que escriba su número de cuenta. Un ATM real también imprime por lo general un recibo al final de una sesión, pero toda la salida de este ATM aparece en la pantalla].

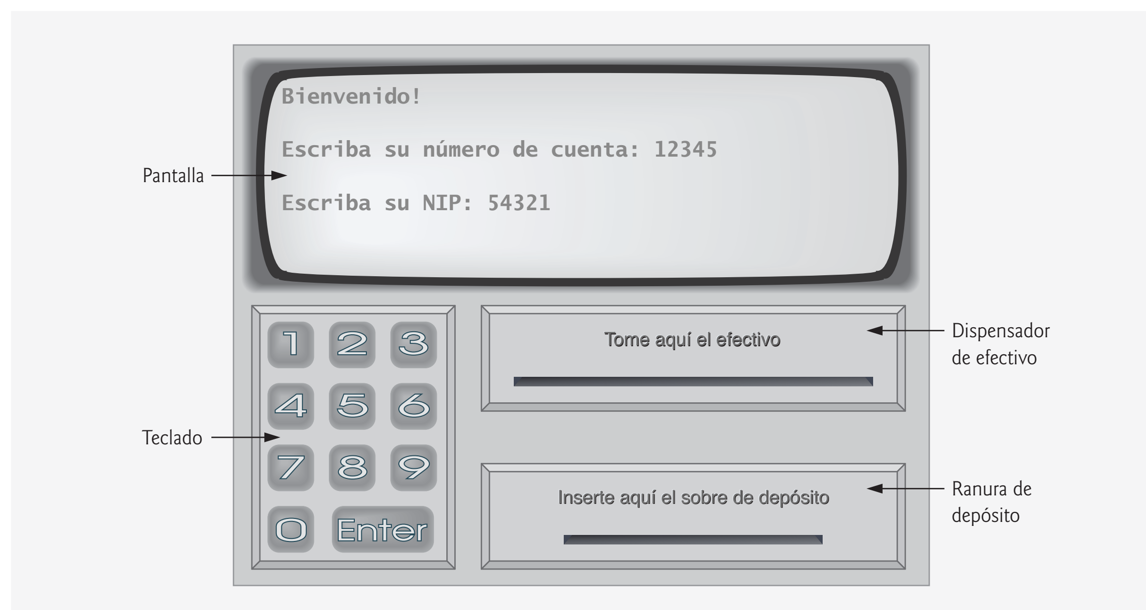


Figura 2.17 | Interfaz de usuario del cajero automático.

El banco desea que usted desarrolle software para realizar las transacciones financieras que inicien los clientes a través del ATM. Posteriormente, el banco integrará el software con el hardware del ATM. El software debe encapsular la funcionalidad de los dispositivos de hardware (por ejemplo: dispensador de efectivo, ranura para depósito) dentro de los componentes de software, pero no necesita estar involucrado en la manera en que estos dispositivos ejecutan su tarea. El hardware del ATM no se ha desarrollado aún, en vez de que usted escriba un software para ejecutarse en el ATM, deberá desarrollar una primera versión del software para que se ejecute en una computadora personal. Esta versión debe utilizar el monitor de la computadora para simular la pantalla del ATM y el teclado de la computadora para simular el teclado numérico del ATM.

Una sesión con el ATM consiste en la autenticación de un usuario (es decir, proporcionar la identidad del usuario) con base en un número de cuenta y un número de identificación personal (NIP), seguida de la creación y la ejecución de transacciones financieras. Para autenticar un usuario y realizar transacciones, el ATM debe interactuar con la base de datos de información sobre las cuentas del banco (es decir, una colección organizada de datos almacenados en una computadora). Para cada cuenta de banco, la base de datos almacena un número de cuenta, un NIP y un saldo que indica la cantidad de dinero en la cuenta. [*Nota:* asumiremos que el banco planea construir sólo un ATM, por lo que no necesitamos preocuparnos para que varios ATMs puedan acceder a esta base de datos al mismo tiempo. Lo que es más, supongamos que el banco no realizará modificaciones en la información que hay en la base de datos mientras un usuario accede al ATM. Además, cualquier sistema comercial como un ATM se topa con cuestiones de seguridad con una complejidad razonable, las cuales van más allá del alcance de un curso de programación de primer o segundo semestre. No obstante, para simplificar nuestro ejemplo supondremos que el banco confía en el ATM para que acceda a la información en la base de datos y la manipule sin necesidad de medidas de seguridad considerables].

Al acercarse al ATM (suponiendo que nadie lo está utilizando), el usuario deberá experimentar la siguiente secuencia de eventos (vea la figura 2.17):

1. La pantalla muestra un mensaje de bienvenida y pide al usuario que introduzca un número de cuenta.
2. El usuario introduce un número de cuenta de cinco dígitos, mediante el uso del teclado.
3. En la pantalla aparece un mensaje, en el que se pide al usuario que introduzca su NIP (número de identificación personal) asociado con el número de cuenta especificado.
4. El usuario introduce un NIP de cinco dígitos mediante el teclado numérico.
5. Si el usuario introduce un número de cuenta válido y el NIP correcto para esa cuenta, la pantalla muestra el menú principal (figura 2.18). Si el usuario introduce un número de cuenta inválido o un NIP incorrecto, la pantalla muestra un mensaje apropiado y después el ATM regresa al *paso 1* para reiniciar el proceso de autenticación.

Una vez que el ATM autentica al usuario, el menú principal (figura 2.18) debe contener una opción numerada para cada uno de los tres tipos de transacciones: solicitud de saldo (opción 1), retiro (opción 2) y depósito (opción 3). El menú principal también debe contener una opción para que el usuario pueda salir del sistema (opción 4). Después el usuario elegirá si desea realizar una transacción (oprimiendo 1, 2 o 3) o salir del sistema (oprimiendo 4).

Si el usuario oprime 1 para solicitar su saldo, la pantalla mostrará el saldo de esa cuenta bancaria. Para ello, el ATM deberá obtener el saldo de la base de datos del banco.

Los siguientes pasos describen las acciones que ocurren cuando el usuario elige la opción 2 para hacer un retiro:

1. La pantalla muestra un menú (vea la figura 2.19) que contiene montos de retiro estándar: $20 (opción 1), $40 (opción 2), $60 (opción 3), $100 (opción 4) y $200 (opción 5). El menú también contiene una opción que permite al usuario cancelar la transacción (opción 6).
2. El usuario introduce la selección del menú mediante el teclado numérico.
3. Si el monto a retirar elegido es mayor que el saldo de la cuenta del usuario, la pantalla muestra un mensaje indicando esta situación y pide al usuario que seleccione un monto más pequeño. Entonces el ATM regresa al *paso 1*. Si el monto a retirar elegido es menor o igual que el saldo de la cuenta del usuario (es decir, un monto de retiro aceptable), el ATM procede al *paso 4*. Si el usuario opta por cancelar la transacción (opción 6), el ATM muestra el menú principal y espera la entrada del usuario.



Figura 2.18 | Menú principal del ATM.



Figura 2.19 | Menú de retiro del ATM.

4. Si el dispensador contiene suficiente efectivo para satisfacer la solicitud, el ATM procede al *paso 5*. En caso contrario, la pantalla muestra un mensaje indicando el problema y pide al usuario que seleccione un monto de retiro más pequeño. Después el ATM regresa al *paso 1*.
5. El ATM carga el monto de retiro al saldo de la cuenta del usuario en la base de datos del banco (es decir, resta el monto de retiro al saldo de la cuenta del usuario).
6. El dispensador de efectivo entrega el monto deseado de dinero al usuario.

7. La pantalla muestra un mensaje para recordar al usuario que tome el dinero.

Los siguientes pasos describen las acciones que ocurren cuando el usuario elige la opción 3 para hacer un depósito:

1. La pantalla muestra un mensaje que pide al usuario que introduzca un monto de depósito o que escriba 0 (cero) para cancelar la transacción.
2. El usuario introduce un monto de depósito o 0 mediante el teclado numérico. [*Nota:* el teclado no contiene un punto decimal o signo de dólares, por lo que el usuario no puede escribir una cantidad real en dólares (por ejemplo, $1.25), sino que debe escribir un monto de depósito en forma de número de centavos (por ejemplo, 125). Después, el ATM divide este número entre 100 para obtener un número que represente un monto en dólares (por ejemplo, 125 ÷ 100 = 1.25)].
3. Si el usuario especifica un monto a depositar, el ATM procede al *paso 4*. Si elije cancelar la transacción (escribiendo 0), el ATM muestra el menú principal y espera la entrada del usuario.
4. La pantalla muestra un mensaje indicando al usuario que introduzca un sobre de depósito en la ranura para depósitos.
5. Si la ranura de depósitos recibe un sobre dentro de un plazo de tiempo no mayor a 2 minutos, el ATM abona el monto del depósito al saldo de la cuenta del usuario en la base de datos del banco (es decir, suma el monto del depósito al saldo de la cuenta del usuario). [*Nota:* este dinero no está disponible de inmediato para retirarse. El banco debe primero verificar físicamente el monto de efectivo en el sobre de depósito, y cualquier cheque que éste contenga debe validarse (es decir, el dinero debe transferirse de la cuenta del emisor del cheque a la cuenta del beneficiario). Cuando ocurra uno de estos eventos, el banco actualizará de manera apropiada el saldo del usuario que está almacenado en su base de datos. Esto ocurre de manera independiente al sistema ATM]. Si la ranura de depósito no recibe un sobre dentro de un plazo de tiempo no mayor a dos minutos, la pantalla muestra un mensaje indicando que el sistema canceló la transacción debido a la inactividad. Después el ATM muestra el menú principal y espera la entrada del usuario.

Una vez que el sistema ejecuta una transacción en forma exitosa, debe volver a mostrar el menú principal para que el usuario pueda realizar transacciones adicionales. Si el usuario elije salir del sistema, la pantalla debe mostrar un mensaje de agradecimiento y después el mensaje de bienvenida para el siguiente usuario.

Análisis del sistema de ATM

En la declaración anterior se presentó un ejemplo simplificado de un documento de requerimientos. Por lo general, dicho documento es el resultado de un proceso detallado de **recopilación de requerimientos**, el cual podría incluir entrevistas con usuarios potenciales del sistema y especialistas en campos relacionados con el mismo. Por ejemplo, un analista de sistemas que se contrate para preparar un documento de requerimientos para software bancario (por ejemplo, el sistema ATM que describimos aquí) podría entrevistar expertos financieros para obtener una mejor comprensión de qué es lo que debe hacer el software. El analista utilizaría la información recopilada para compilar una lista de **requerimientos del sistema**, para guiar a los diseñadores de sistemas en el proceso de diseño del mismo.

El proceso de recopilación de requerimientos es una tarea clave de la primera etapa del ciclo de vida del software. El **ciclo de vida del software** especifica las etapas a través de las cuales el software evoluciona desde el momento en que fue concebido hasta que deja de utilizarse. Por lo general, estas etapas incluyen: análisis, diseño, implementación, prueba y depuración, despliegue, mantenimiento y retiro. Existen varios modelos de ciclo de vida del software, cada uno con sus propias preferencias y especificaciones con respecto a cuándo y qué tan a menudo deben llevar a cabo los ingenieros de software las diversas etapas. Los **modelos de cascada** realizan cada etapa una vez en sucesión, mientras que los **modelos iterativos** pueden repetir una o más etapas varias veces a lo largo del ciclo de vida de un producto.

La etapa de análisis del ciclo de vida del software se enfoca en definir el problema a resolver. Al diseñar cualquier sistema, uno debe *resolver el problema de la manera correcta*, pero de igual manera uno debe *resolver el problema correcto*. Los analistas de sistemas recolectan los requerimientos que indican el problema específico a resolver. Nuestro documento de requerimientos describe nuestro sistema ATM con el suficiente detalle como para que usted no necesite pasar por una etapa de análisis exhaustiva; ya lo hicimos por usted.

Para capturar lo que debe hacer un sistema propuesto, los desarrolladores emplean a menudo una técnica conocida como **modelado de caso-uso**. Este proceso identifica los **casos de uso** del sistema, cada uno de los cuales representa una capacidad distinta que el sistema provee a sus clientes. Por ejemplo, es común que los ATMs tengan varios casos de uso, como "Ver saldo de cuenta", "Retirar efectivo", "Depositar fondos", "Transferir fondos entre cuentas" y "Comprar estampas postales". El sistema ATM simplificado que construiremos en este ejemplo práctico requiere sólo los tres primeros casos de uso.

Cada uno de los casos de uso describe un escenario común en el cual el usuario utiliza el sistema. Usted ya leyó las descripciones de los casos de uso del sistema ATM en el documento de requerimientos; las listas de pasos requeridos para realizar cada tipo de transacción (como solicitud de saldo, retiro y depósito) describen en realidad los tres casos de uso de nuestro ATM: "Ver saldo de cuenta", "Retirar efectivo" y "Depositar fondos", respectivamente.

Diagramas de caso-uso

Ahora presentaremos el primero de varios diagramas de UML en el ejemplo práctico. Crearemos un **diagrama de caso-uso** para modelar las interacciones entre los clientes de un sistema (en este ejemplo práctico, los clientes del banco) y sus casos de uso. El objetivo es mostrar los tipos de interacciones que tienen los usuarios con un sistema sin proveer los detalles; éstos se mostrarán en otros diagramas de UML (los cuales presentaremos a lo largo del ejemplo práctico). A menudo, los diagramas de caso-uso se acompañan de texto informal que describe los casos de uso con más detalle; como el texto que aparece en el documento de requerimientos. Los diagramas de caso-uso se producen durante la etapa de análisis del ciclo de vida del software. En sistemas más grandes, los diagramas de caso-uso son herramientas indispensables que ayudan a los diseñadores de sistemas a enfocarse en satisfacer las necesidades de los usuarios.

La figura 2.20 muestra el diagrama de caso-uso para nuestro sistema ATM. La figura humana representa a un **actor**, el cual define los roles que desempeña una entidad externa (como una persona u otro sistema) cuando interactúa con el sistema. Para nuestro cajero automático, el actor es un Usuario que puede ver el saldo de una cuenta, retirar efectivo y depositar fondos del ATM. El Usuario no es una persona real, sino que constituye los roles que puede desempeñar una persona real (al desempeñar el papel de un Usuario) mientras interactúa con el ATM. Hay que tener en cuenta que un diagrama de caso-uso puede incluir varios actores. Por ejemplo, el diagrama de caso-uso para un sistema ATM de un banco real podría incluir también un actor llamado Administrador, que rellene el dispensador de efectivo a diario.

Nuestro documento de requerimientos provee los actores: "los usuarios del ATM deben poder ver el saldo de su cuenta, retirar efectivo y depositar fondos". Por lo tanto, el actor en cada uno de estos tres casos de uso es el usuario que interactúa con el ATM. Una entidad externa (una persona real) desempeña el papel del usuario para realizar transacciones financieras. La figura 2.20 muestra un actor, cuyo nombre (Usuario) aparece debajo del actor en el diagrama. UML modela cada caso de uso como un óvalo conectado a un actor con una línea sólida.

Los ingenieros de software (más específicamente, los diseñadores de sistemas) deben analizar el documento de requerimientos o un conjunto de casos de uso, y diseñar el sistema antes de que los programadores lo implementen en un lenguaje de programación específico. Durante la etapa de análisis, los diseñadores de sistemas se enfocan en comprender el documento de requerimientos para producir una especificación de alto nivel que describa *qué* es lo que el sistema debe hacer. El resultado de la etapa de diseño (una **especificación de diseño**)

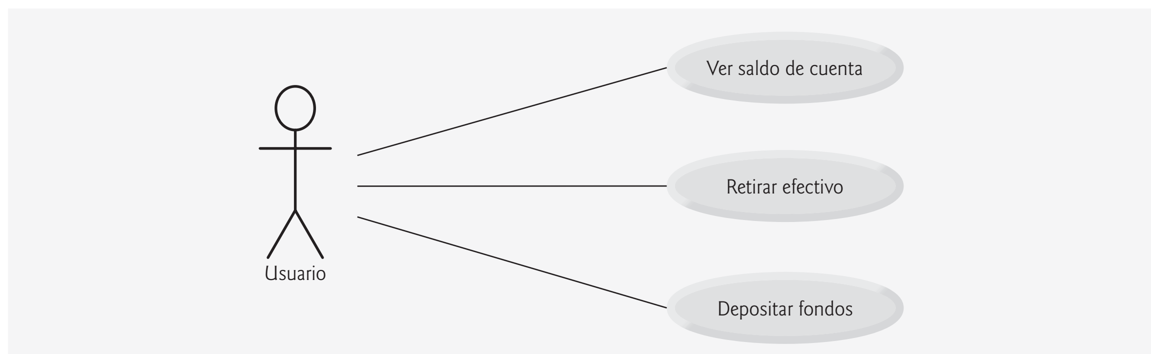


Figura 2.20 | Diagrama de caso-uso para el sistema ATM, desde la perspectiva del usuario.

debe detallar claramente *cómo* debe construirse el sistema para satisfacer estos requerimientos. En las siguientes secciones del Ejemplo práctico de Ingeniería de Software, llevaremos a cabo los pasos de un proceso simple de diseño orientado a objetos (DOO) con el sistema ATM, para producir una especificación de diseño que contenga una colección de diagramas de UML y texto de apoyo. UML está diseñado para utilizarse con cualquier proceso de DOO. Existen muchos de esos procesos, de los cuales el más conocido es Rational Unified Process™ (RUP), desarrollado por Rational Software Corporation. RUP es un proceso robusto para diseñar aplicaciones a nivel industrial. Para este ejemplo práctico, presentaremos nuestro propio proceso de diseño simplificado, desarrollado para estudiantes de cursos de programación de primer y segundo semestre.

Diseño del sistema ATM

Ahora comenzaremos la etapa de diseño de nuestro sistema ATM. Un **sistema** es un conjunto de componentes que interactúan para resolver un problema. Por ejemplo, para realizar sus tareas designadas, nuestro sistema ATM tiene una interfaz de usuario (figura 2.17), contiene software para ejecutar transacciones financieras e interactúa con una base de datos de información de cuentas bancarias. La **estructura del sistema** describe los objetos del sistema y sus interrelaciones. El **comportamiento del sistema** describe la manera en que cambia el sistema a medida que sus objetos interactúan entre sí. Todo sistema tiene tanto estructura como comportamiento; los diseñadores deben especificar ambos. Existen diversos tipos de estructuras y comportamientos de un sistema. Por ejemplo, las interacciones entre los objetos en el sistema son distintas a las interacciones entre el usuario y el sistema, pero aun así ambas constituyen una porción del comportamiento del sistema.

El estándar UML 2 especifica 13 tipos de diagramas para documentar los modelos de un sistema. Cada tipo de diagrama modela una característica distinta de la estructura o del comportamiento de un sistema; seis diagramas se relacionan con la estructura del sistema; los siete restantes se relacionan con su comportamiento. Aquí listaremos sólo los seis tipos de diagramas que utilizaremos en nuestro ejemplo práctico, uno de los cuales (el diagrama de clases) modela la estructura del sistema, mientras que los otros cinco modelan el comportamiento. En el apéndice O, UML 2: Tipos de diagramas adicionales, veremos las generalidades sobre los siete tipos restantes de diagramas de UML.

1. Los **diagramas de caso-uso**, como el de la figura 2.20, modelan las interacciones entre un sistema y sus entidades externas (actores) en términos de casos de uso (capacidades del sistema, como "Ver saldo de cuenta", "Retirar efectivo" y "Depositar fondos").
2. Los **diagramas de clases**, que estudiará en la sección 3.10, modelan las clases o "bloques de construcción" que se utilizan en un sistema. Cada sustantivo u "objeto" que se describe en el documento de requerimientos es candidato para ser una clase en el sistema (por ejemplo, `Cuenta`, `Teclado`). Los diagramas de clases nos ayudan a especificar las relaciones estructurales entre las partes del sistema. Por ejemplo, el diagrama de clases del sistema ATM especificará que el ATM está compuesto físicamente de una pantalla, un teclado, un dispensador de efectivo y una ranura para depósitos.
3. Los **diagramas de máquina de estado**, que estudiará en la sección 5.11, modelan las formas en que un objeto cambia de estado. El **estado** de un objeto se indica mediante los valores de todos los atributos del objeto, en un momento dado. Cuando un objeto cambia de estado, puede comportarse de manera distinta en el sistema. Por ejemplo, después de validar el NIP de un usuario, el ATM cambia del estado "usuario no autenticado" al estado "usuario autenticado", punto en el cual el ATM permite al usuario realizar transacciones financieras (por ejemplo, ver el saldo de su cuenta, retirar efectivo, depositar fondos).
4. Los **diagramas de actividad**, que también estudiará en la sección 5.11, modelan la **actividad** de un objeto: el flujo de trabajo (secuencia de eventos) del objeto durante la ejecución del programa. Un diagrama de actividad modela las acciones que realiza el objeto y especifica el orden en el cual desempeña estas acciones. Por ejemplo, un diagrama de actividad muestra que el ATM debe obtener el saldo de la cuenta del usuario (de la base de datos de información de las cuentas del banco) antes de que la pantalla pueda mostrar el saldo al usuario.
5. Los **diagramas de comunicación** (llamados **diagramas de colaboración** en versiones anteriores de UML) modelan las interacciones entre los objetos en un sistema, con un énfasis acerca de *qué* interacciones ocurren. En la sección 7.14 aprenderá que estos diagramas muestran cuáles objetos deben interactuar para realizar una transacción en el ATM. Por ejemplo, el ATM debe comunicarse con la base de datos de información de las cuentas del banco para obtener el saldo de una cuenta.

6. Los **diagramas de secuencia** modelan también las interacciones entre los objetos en un sistema, pero a diferencia de los diagramas de comunicación, enfatizan *cuándo* ocurren las interacciones. En la sección 7.14 aprenderá que estos diagramas ayudan a mostrar el orden en el que ocurren las interacciones al ejecutar una transacción financiera. Por ejemplo, la pantalla pide al usuario que escriba un monto de retiro antes de dispensar el efectivo.

En la sección 3.10 seguiremos diseñando nuestro sistema ATM; ahí identificaremos las clases del documento de requerimientos. Para lograr esto, extraeremos sustantivos clave y frases nominales del documento de requerimientos. Mediante el uso de estas clases, desarrollaremos nuestro primer borrador del diagrama de clases que modelará la estructura de nuestro sistema ATM.

Recursos en Internet y Web

Los siguientes URLs proporcionan información sobre el diseño orientado a objetos con UML.

`www-306.ibm.com/software/rational/uml/`

Lista preguntas frecuentes acerca del UML, proporcionado por IBM Rational.

`www.douglass.co.uk/documents/softdocwiz.com.UML.htm`

Sitio anfitrión del Diccionario del Lenguaje unificado de modelado, el cual lista y define todos los términos utilizados en el UML.

`www-306.ibm.com/software/rational/offerings/design.html`

Proporciona información acerca del software IBM Rational, disponible para el diseño de sistemas. Ofrece descargas de versiones de prueba de 30 días de varios productos, como IBM Rational Rose® XDE Developer.

`www.embarcadero.com/products/describe/index.html`

Proporciona una licencia gratuita de 14 días para descargar una versión de prueba de Describe™: una herramienta de modelado con UML de Embarcadero Technologies®.

`www.borland.com/us/products/together/index.html`

Proporciona una licencia gratuita de 30 días para descargar una versión de prueba de Borland® Together® Control Center™: una herramienta de desarrollo de software que soporta el UML.

`www.ilogix.com/sublevel.aspx?id=53 http://modelingcommunity.telelogic.com/developer-trial.aspx`

Proporciona una licencia gratuita de 30 días para descargar una versión de prueba de I-Logix Rhapsody®: un entorno de desarrollo controlado por modelos y basado en UML 2.

`argouml.tigris.org`

Contiene información y descargas para ArgoUML, una herramienta gratuita de código fuente abierto de UML, escrita en Java.

`www.objectsbydesign.com/books/booklist.html`

Provee una lista de libros acerca de UML y el diseño orientado a objetos.

`www.objectsbydesign.com/tools/umltools_byCompany.html`

Provee una lista de herramientas de software que utilizan UML, como IBM Rational Rose, Embarcadero Describe, Sparx Systems Enterprise Architect, I-Logix Rhapsody y Gentleware Poseidon para UML.

`www.ootips.org/ood-principles.html`

Proporciona respuestas a la pregunta "¿Qué se requiere para tener un buen diseño orientado a objetos?"

`parlezuml.com/tutorials/umlforjava.htm`

Ofrece un tutorial de UML para desarrolladores de Java, el cual presenta los diagramas de UML y los compara detalladamente con el código que los implementa.

`www.cetus-links.org/oo_uml.html`

Introduce el UML y proporciona vínculos a numerosos recursos sobre UML.

`www.agilemodeling.com/essays/umlDiagrams.htm`

Proporciona descripciones detalladas y tutoriales acerca de cada uno de los 13 tipos de diagramas de UML 2.

Lecturas recomendadas

Los siguientes libros proporcionan información acerca del diseño orientado a objetos con UML.

Booch, G. *Object-Oriented Analysis and Design with Applications, Tercera edición*. Boston: Addison-Wesley, 2004.

Eriksson, H. *et al. UML 2 Toolkit*. Nueva York: John Wiley, 2003.

Kruchten, P. *The Rational Unified Process: An Introduction*. Boston: Addison-Wesley, 2004.

Larman, C. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design,* Segunda edición. Upper Saddle River, NJ: Prentice Hall, 2002.

Roques, P. *UML in Practice: The Art of Modeling Software Systems Demonstrated Through Worked Examples and Solutions*. Nueva York: John Wiley, 2004.

Rosenberg, D. y K. Scott. *Applying Use Case Driven Object Modeling with UML: An Annotated e-Commerce Example*. Reading, MA: Addison-Wesley, 2001.

Rumbaugh, J., I. Jacobson y G. Booch. *The Complete UML Training Course.* Upper Saddle River, NJ: Prentice Hall, 2000.

Rumbaugh, J., I. Jacobson y G. Booch. *The Unified Modeling Language Reference Manual*. Reading, MA: Addison-Wesley, 1999.

Rumbaugh, J., I. Jacobson y G. Booch. *The Unified Software Development Process*. Reading, MA: Addison-Wesley, 1999.

Ejercicios de autoevaluación del Ejemplo práctico de Ingeniería de Software

2.1 Suponga que habilitamos a un usuario de nuestro sistema ATM para transferir dinero entre dos cuentas bancarias. Modifique el diagrama de caso-uso de la figura 2.20 para reflejar este cambio.

2.2 Los __________ modelan las interacciones entre los objetos en un sistema, con énfasis acerca de *cuándo* ocurren estas interacciones.

a) Diagramas de clases
b) Diagramas de secuencia
c) Diagramas de comunicación
d) Diagramas de actividad

2.3 ¿Cuál de las siguientes opciones lista las etapas de un típico ciclo de vida de software, en orden secuencial?

a) diseño, análisis, implementación, prueba
b) diseño, análisis, prueba, implementación
c) análisis, diseño, prueba, implementación
d) análisis, diseño, implementación, prueba

Respuestas a los ejercicios de autoevaluación del Ejemplo práctico de Ingeniería de Software

2.1 La figura 2.21 contiene un diagrama de caso-uso para una versión modificada de nuestro sistema ATM, que también permite a los usuarios transferir dinero entre cuentas.

2.2 b.

2.3 d.

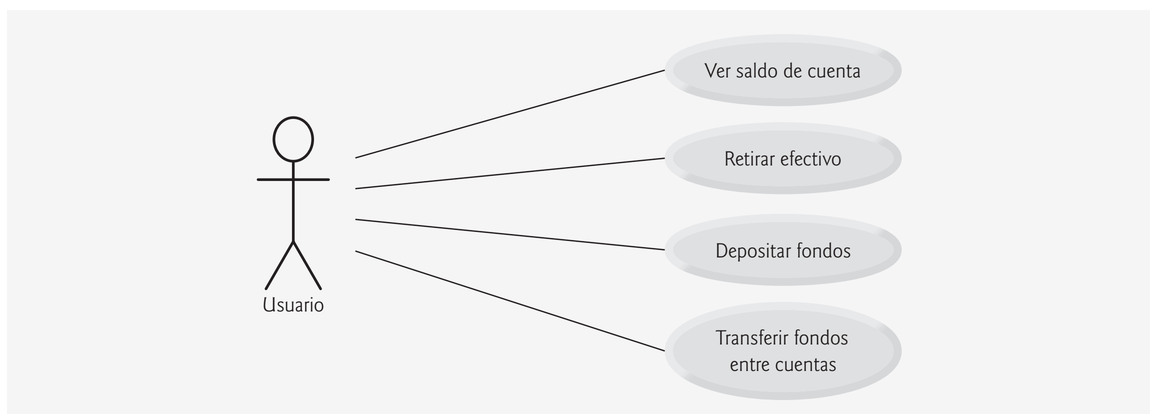


Figura 2.21 | Diagrama de caso-uso para una versión modificada de nuestro sistema ATM, que también permite a los usuarios transferir dinero entre varias cuentas.

2.10 Conclusión

En este capítulo aprendió muchas características importantes de Java, incluyendo cómo mostrar datos en la pantalla en un Símbolo del sistema, recibir datos del teclado, realizar cálculos y tomar decisiones. Las aplicaciones que presentamos aquí le sirvieron como una introducción a los conceptos básicos de programación. Como verá en el capítulo 3, por lo general las aplicaciones de Java contienen sólo unas cuantas líneas de código en el método `main`; comúnmente estas instrucciones crean los objetos que realizan el trabajo de la aplicación. En el capítulo 3 aprenderá a implementar sus propias clases, y a utilizar objetos de esas clases en las aplicaciones.

Resumen

Sección 2.2 Su primer programa en Java: imprimir una línea de texto

- Los programadores de computadoras crean aplicaciones, escribiendo programas de cómputo. Una aplicación de Java es un programa de computadora que se ejecuta cuando utilizamos el comando `java` para iniciar la JVM.
- Los programadores insertan comentarios para documentar los programas y mejorar su legibilidad. El compilador de Java ignora los comentarios.
- Un comentario que empieza con `//` se llama comentario de fin de línea (o de una sola línea), ya que termina al final de la línea en la que aparece.
- Los comentarios tradicionales (de varias líneas) pueden dividirse en varias líneas, y están delimitados por `/*` y `*/`. El compilador ignora todo el texto entre los delimitadores.
- Los comentarios Javadoc se delimitan por `/**` y `*/`. Estos comentarios permiten a los programadores incrustar la documentación directamente en sus programas. La herramienta `javadoc` genera documentación en HTML, con base en los comentarios Javadoc.
- La sintaxis de un lenguaje de programación especifica las reglas para crear un programa apropiado en ese lenguaje.
- Un error de sintaxis (también conocido como error de compilador, error en tiempo de compilación o error de compilación) ocurre cuando el compilador encuentra código que viola las reglas del lenguaje Java.
- Los programadores utilizan líneas en blanco y espacios para facilitar la lectura de los programas. En conjunto, las líneas en blanco, los espacios y los tabuladores se conocen como espacio en blanco. Los espacios y los tabuladores se conocen específicamente como caracteres de espacio en blanco. El compilador ignora el espacio en blanco.
- Todo programa en Java consiste en por lo menos una declaración de clase, definida por el programador (también conocida como clase definida por el programador, o clase definida por el usuario).
- Las palabras clave están reservadas para el uso exclusivo de Java, y siempre se escriben con letras minúsculas.
- La palabra clave `class` introduce una declaración de clase, y va seguida inmediatamente del nombre de la clase.
- Por convención, todos los nombres de las clases en Java empiezan con una letra mayúscula, y la primera letra de cada palabra subsiguiente también se escribe en mayúscula (como `NombreClaseDeEjemplo`).
- El nombre de una clase de Java es un identificador: una serie de caracteres formada por letras, dígitos, guiones bajos (`_`) y signos de dólar (`$`), que no empieza con un dígito y no contiene espacios. Por lo general, un identificador que no empieza con letra mayúscula no es el nombre de una clase de Java.
- Java es sensible a mayúsculas/minúsculas; es decir, las letras mayúsculas y minúsculas son distintas.
- El cuerpo de todas las declaraciones de clases debe estar delimitado por llaves, `{` y `}`.
- La declaración de una clase `public` debe guardarse en un archivo con el mismo nombre que la clase, seguido de la extensión de nombre de archivo "`.java`".
- El método `main` es el punto de inicio de toda aplicación en Java, y debe empezar con:

  ```
  public static void main( String args[] )
  ```

  en caso contrario, la JVM no ejecutará la aplicación.
- Los métodos pueden realizar tareas y devolver información cuando completan éstas tareas. La palabra clave `void` indica que un método realizará una tarea, pero no devolverá información.
- Las instrucciones instruyen a la computadora para que realice acciones.
- Una secuencia de caracteres entre comillas dobles se llama cadena, cadena de caracteres, mensaje o literal de cadena.
- `System.out` es el objeto de salida estándar; permite a las aplicaciones de Java mostrar caracteres en la ventana de comandos.
- El método `System.out.println` muestra su argumento en la ventana de comandos, seguido de un carácter de nueva línea para colocar el cursor de salida en el inicio de la siguiente línea.

- Toda instrucción termina con un punto y coma.
- La mayoría de los sistemas operativos utilizan el comando `cd` para cambiar directorios en la ventana de comandos.
- Para compilar un programa se utiliza el comando `javac`. Si el programa no contiene errores de sintaxis, se crea un archivo de clase que contiene los códigos de bytes de Java, los cuales representan a la aplicación. La JVM interpreta estos códigos de bytes cuando ejecutamos el programa.

Sección 2.3 Modificación de nuestro primer programa en Java

- `System.out.print` muestra su argumento en pantalla y coloca el cursor de salida justo después del último carácter visualizado.
- Una barra diagonal inversa (`\`) en una cadena es un carácter de escape. Indica que se va a imprimir un "carácter especial". Java combina el siguiente carácter con la barra diagonal inversa para formar una secuencia de escape. La secuencia de escape `\n` representa el carácter de nueva línea, el cual coloca el cursor en la siguiente línea.

Sección 2.4 Cómo mostrar texto con `printf`

- El método `System.out.printf` (`f` significa "formato") muestra datos con formato.
- Cuando un método requiere varios argumentos, éstos se separan con comas (`,`); a esto se le conoce como lista separada por comas.
- El primer argumento del método `printf` es una cadena de formato, que puede consistir en texto fijo y especificadores de formato. El método `printf` imprime el texto fijo de igual forma que `print` o `println`. Cada especificador de formato es un receptáculo para un valor, y especifica el tipo de datos a imprimir.
- Los especificadores de formato empiezan con un signo porcentual (`%`), y van seguidos de un carácter que representa el tipo de datos. El especificador de formato `%s` es un receptáculo para una cadena.
- En la posición del primer especificador de formato, `printf` sustituye el valor del primer argumento después de la cadena de formato. En la posición de los siguientes especificadores de formato, `printf` sustituye el valor del siguiente argumento en la lista de argumentos.

Sección 2.5 Otra aplicación en Java: suma de enteros

- Los enteros son números completos, como –22 ,7, 0 y 1024.
- Una declaración `import` ayuda al compilador a localizar una clase que se utiliza en un programa.
- Java cuenta con un extenso conjunto de clases predefinidas que los programadores pueden reutilizar, en vez de tener que "reinventar la rueda". Estas clases se agrupan en paquetes: llamados colecciones de clases.
- En conjunto, a los paquetes de Java se les conoce como la biblioteca de clases de Java, o la Interfaz de Programación de Aplicaciones de Java (API de Java).
- Una instrucción de declaración de variable especifica el nombre y el tipo de una variable.
- Una variable es una ubicación en la memoria de la computadora, en la cual se puede guardar un valor para usarlo posteriormente en un programa. Todas las variables deben declararse con un nombre y un tipo para poder utilizarlas.
- El nombre de una variable permite al programa acceder al valor de la variable en memoria. Un nombre de variable puede ser cualquier identificador válido.
- Al igual que otras instrucciones, las instrucciones de declaración de variables terminan con un punto y coma (`;`).
- Un objeto `Scanner` (paquete `java.util`) permite a un programa leer datos para usarlos en éste. Los datos pueden provenir de muchas fuentes, como un archivo en disco o del usuario, a través del teclado. Antes de usar un objeto `Scanner`, el programa debe crearlo y especificar el origen de los datos.
- Las variables deben inicializarse para poder usarlas en un programa.
- La expresión `new Scanner( System.in )` crea un objeto `Scanner` que lee datos desde el teclado. El objeto de entrada estándar, `System.in`, permite a las aplicaciones de Java leer los datos escritos por el usuario.
- El tipo de datos `int` se utiliza para declarar variables que guardarán valores enteros. El rango de valores para un `int` es de –2,147,483,648 a +2,147,483,647.
- Los tipos `float` y `double` especifican números reales, y el tipo `char` especifica datos de caracteres. Los números reales son números que contienen puntos decimales, como `3.4`, `0.0` y `-11.19`. Las variables de tipo `char` representan caracteres individuales, como una letra mayúscula (por ejemplo, `A`), un dígito (por ejemplo, `7`), un carácter especial (por ejemplo, `*` o `%`) o una secuencia de escape (por ejemplo, el carácter de nueva línea, `\n`).
- Los tipos como `int`, `float`, `double` y `char` se conocen comúnmente como tipos primitivos o tipos integrados. Los nombres de los tipos primitivos son palabras clave; por ende, deben aparecer escritos sólo con letras minúsculas.
- Un indicador pide al usuario que realice una acción específica.
- El método `nextInt` de `Scanner` obtiene un entero para usarlo en un programa.

- El operador de asignación, =, permite al programa dar un valor a una variable. El operador = se llama operador binario, ya que tiene dos operandos. Una instrucción de asignación utiliza un operador de asignación para asignar un valor a una variable.
- Las partes de las instrucciones que tienen valores se llaman expresiones.
- El especificador de formato `%d` es un receptáculo para un valor `int`.

Sección 2.6 Conceptos acerca de la memoria

- Los nombres de las variables corresponden a ubicaciones en la memoria de la computadora. Cada variable tiene un nombre, un tipo, un tamaño y un valor.
- Cada vez que se coloca un valor en una ubicación de memoria, se sustituye al valor anterior en esa ubicación. El valor anterior se pierde.

Sección 2.7 Aritmética

- La mayoría de los programas realizan cálculos aritméticos. Los operadores aritméticos son + (suma), – (resta), * (multiplicación), / (división) y % (residuo).
- La división de enteros produce un cociente entero.
- El operador residuo, %, produce el residuo después de la división.
- Las expresiones aritméticas en Java deben escribirse en formato de línea recta.
- Si una expresión contiene paréntesis anidados, el conjunto de paréntesis más interno se evalúa primero.
- Java aplica los operadores en las expresiones aritméticas en una secuencia precisa, la cual se determina mediante las reglas de precedencia de los operadores.
- Cuando decimos que los operadores se aplican de izquierda a derecha, nos referimos a su asociatividad. Algunos operadores se asocian de derecha a izquierda.
- Los paréntesis redundantes en una expresión pueden hacer que ésta sea más clara.

Sección 2.8 Toma de decisiones: operadores de igualdad y relacionales

- Una condición es una expresión que puede ser verdadera o falsa. La instrucción `if` de Java permite que un programa tome una decisión, con base en el valor de una condición.
- Las condiciones en las instrucciones `if` se forman mediante el uso de los operadores de igualdad (`==` y `!=`) y relacionales (`>`, `<`, `>=` y `<=`).
- Una instrucción `if` siempre empieza con la palabra clave `if`, seguida de una condición entre paréntesis, y espera una instrucción en su cuerpo.
- La instrucción vacía es una instrucción que no realiza una tarea.

Terminología

`%d`, especificador de formato
`%s`, especificador de formato
`.class`, extensión de archivo
`.java`, extensión de archivo
aplicación
archivo de clase
argumento
asociatividad de los operadores
autodocumentado
barra diagonal inversa (\), carácter de escape
biblioteca de clases de Java
cadena
cadena de caracteres
cadena de formato
carácter de escape
caracteres de espacio en blanco
`cd`, comando para cambiar directorios
`char`, tipo primitivo
clase definida por el programador
clase definida por el usuario
`class`, palabra clave
comentario
comentario de fin de línea (`//`)
comentario de una sola línea (`//`)
comentario de varias líneas (`/* */`)
comentario tradicional (`/* */`)
condición
cuerpo de la declaración de un método
cuerpo de la declaración de una clase
cursor de salida
decisión
declaración de una clase
declaración de variable
división de enteros
división, operador (`/`)
documentación de la API de Java
documento de un programa
`double`, tipo primitivo
entero
error de compilación
error de compilador
error de sintaxis

error en tiempo de compilación
espacio en blanco
especificador de formato
false
float, tipo primitivo
formato de línea recta
identificador
if, instrucción
igualdad, operadores
  == "es igual a"
  != "no es igual a"
import, declaración
indicador
instrucción
instrucción de asignación
instrucción de declaración de variable
instrucción vacía (;)
int (entero), tipo primitivo
Interfaz de Programación de Aplicaciones (API) de Java
java, comando
java.lang, paquete
Javadoc, comentario (/** */)
javadoc, programa de utilería
línea de comandos
lista separada por comas
literal de cadena
llave derecha (})
llave izquierda ({)
main, método
mensaje
método
multiplicación, operador (*)
nombre de una clase
nombre de una variable
nombre de variable
nueva línea, carácter (\n)
objeto
objeto de entrada estándar (System.in)
objeto de salida estándar (System.out)
operador
operador binario
operador de asignación (=)
operador de suma (+)
operadores aritméticos (*, /, %, + y –)
operando
palabras reservadas
paquete
paréntesis ()
paréntesis anidados
paréntesis redundantes
precedencia
programa de cómputo
public, palabra clave
punto y coma (;)
realizar una acción
reglas de precedencia de operadores
relacionales, operadores
  < "es menor que"
  <= "es menor o igual a"
  > "es mayor que"
  >= "es mayor o igual a"
residuo, operador (%)
resta, operador (–)
Scanner, clase
secuencia de escape
sensible a mayúsculas/minúsculas
shell
símbolo de MS-DOS
Símbolo del sistema
sintaxis
System.in, objeto (entrada estándar)
System.out, objeto (salida estándar)
System.out.print, método
System.out.printf, método
System.out.println, método
tamaño de una variable
texto fijo en una cadena de formato
tipo de una variable
tipo integrado
tipo primitivo
tolerante a fallas
true
ubicación de memoria
ubicación de una variable
valor de variable
variable
ventana de comandos
ventana de Terminal
void, palabra clave

Ejercicios de autoevaluación

2.1 Complete las siguientes oraciones:

a) El cuerpo de cualquier método comienza con un(a) ______________ y termina con un(a) ______________.
b) Toda instrucción termina con un ______________.
c) La instrucción ______________ (presentada en este capítulo) se utiliza para tomar decisiones.
d) ______________ indica el inicio de un comentario de fin de línea.
e) ______________, ______________, ______________ y ______________ se conocen como espacio en blanco.

f) Las ______________ están reservadas para su uso en Java.
g) Las aplicaciones en Java comienzan su ejecución en el método ______________.
h) Los métodos ______________, ______________ y ______________ muestran información en la ventana de comandos.

2.2 Indique si cada una de las siguientes instrucciones es *verdadera* o *falsa*. Si es *falsa*, explique por qué.
a) Los comentarios hacen que la computadora imprima el texto que va después de los caracteres `//` en la pantalla, al ejecutarse el programa.
b) Todas las variables deben recibir un tipo cuando se declaran.
c) Java considera que las variables `numero` y `NuMeRo` son idénticas.
d) El operador residuo (`%`) puede utilizarse solamente con operandos enteros.
e) Los operadores aritméticos `*`, `/`, `%`, `+` y `–` tienen todos el mismo nivel de precedencia.

2.3 Escriba instrucciones para realizar cada una de las siguientes tareas:
a) Declarar las variables `c`, `estaEsUnaVariable`, `q76354` y `numero` como de tipo `int`.
b) Pedir al usuario que introduzca un entero.
c) Recibir un entero como entrada y asignar el resultado a la variable `int valor`. Suponga que se puede utilizar la variable `entrada` tipo `Scanner` para recibir un valor del teclado.
d) Si la variable `numero` no es igual a `7`, mostrar `"La variable numero no es igual a 7"`.
e) Imprimir `"Este es un programa en Java"` en una línea de la ventana de comandos.
f) Imprimir `"Este es un programa en Java"` en dos líneas de la ventana de comandos. La primera línea debe terminar con `es un`. Use el método `System.out.println`.
g) Imprimir `"Este es un programa en Java"` en dos líneas de la ventana de comandos. La primera línea debe terminar con `es un`. Use el método `System.out.printf` y dos especificadores de formato `%s`.

2.4 Identifique y corrija los errores en cada una de las siguientes instrucciones:
a)
```
if ( c < 7 );
    System.out.println( "c es menor que 7" );
```
b)
```
if ( c => 7 )
    System.out.println( "c es igual o mayor que 7" );
```

2.5 Escriba declaraciones, instrucciones o comentarios para realizar cada una de las siguientes tareas:
a) Indicar que un programa calculará el producto de tres enteros.
b) Crear un objeto `Scanner` que lea valores de la entrada estándar.
c) Declarar las variables `x`, `y`, `z` y `resultado` de tipo `int`.
d) Pedir al usuario que escriba el primer entero.
e) Leer el primer entero del usuario y almacenarlo en la variable `x`.
f) Pedir al usuario que escriba el segundo entero.
g) Leer el segundo entero del usuario y almacenarlo en la variable `y`.
h) Pedir al usuario que escriba el tercer entero.
i) Leer el tercer entero del usuario y almacenarlo en la variable `z`.
j) Calcular el producto de los tres enteros contenidos en las variables `x`, `y` y `z`, y asignar el resultado a la variable `resultado`.
k) Mostrar el mensaje `"El producto es"`, seguido del valor de la variable `resultado`.

2.6 Utilizando las instrucciones que escribió en el ejercicio 2.5, escriba un programa completo que calcule e imprima el producto de tres enteros.

Respuestas a los ejercicios de autoevaluación

2.1 a) llave izquierda (`{`), llave derecha (`}`). b) punto y coma (`;`). c) `if`. d) `//`. e) Líneas en blanco, caracteres de espacio, caracteres de nueva línea y tabuladores. f) palabras clave. g) `main`. h) `System.out.print`, `System.out.println` y `System.out.printf`.

2.2 a) Falso. Los comentarios no producen ninguna acción cuando el programa se ejecuta. Se utilizan para documentar programas y mejorar su legibilidad.
b) Verdadero.
c) Falso. Java es sensible a mayúsculas y minúsculas, por lo que estas variables son distintas.

d) Falso. El operador residuo puede utilizarse también con operandos no enteros en Java.
e) Falso. Los operadores `*`, `/` y `%` se encuentran en el mismo nivel de precedencia, y los operadores `+` y `-` se encuentran en un nivel menor de precedencia.

2.3 a)
```
int c, estaEsUnaVariable, q76354, numero;
```
o
```
int c;
int estaEsUnaVariable;
int q76354;
int numero;
```
b) `System.out.print( "Escriba un entero " );`
c) `valor = entrada.nextInt();`
d)
```
if ( numero != 7 )
System.out.println( "La variable numero no es igual a 7" );
```
e) `System.out.println( "Este es un programa en Java" );`
f) `System.out.println( "Este es un\n programa en Java" );`
g) `System.out.printf( "%s\%s\n", "Este es un", "programa en Java" );`

2.4 Las soluciones al ejercicio de autoevaluación 2.4 son las siguientes:
a) Error: hay un punto y coma después del paréntesis derecho de la condición `(c < 7 )` en la instrucción `if`. Corrección: quite el punto y coma que va después del paréntesis derecho. [*Nota:* como resultado, la instrucción de salida se ejecutará, sin importar que la condición en la instrucción `if` sea verdadera].
b) Error: el operador relacional => es incorrecto. Corrección: cambie => a >=.

2.5 a) `// Calcula el producto de tres enteros`
b) `Scanner entrada = new Scanner (System.in);`
c)
```
int x, y, z, resultado;
```
o
```
int x;
int y;
int z;
int resultado;
```
d) `System.out.print( "Escriba el primer entero: " );`
e) `x = entrada.nextInt();`
f) `System.out.print( "Escriba el segundo entero: " );`
g) `y = entrada.nextInt();`
h) `System.out.print( "Escriba el tercer entero: " );`
i) `z = entrada.nextInt();`
j) `resultado = x * y * z;`
k) `System.out.printf( "El producto es %d\n", resultado );`
l) `System.exit( 0 );`

2.6 La solución para el ejercicio 2.6 es la siguiente:

```
// Ejemplo 2.6: Producto.java
// Calcular el producto de tres enteros.
import java.util.Scanner; // el programa usa Scanner

public class Producto
{
   public static void main( String args[] )
   {
      // crea objeto Scanner para obtener la entrada de la ventana de comandos
      Scanner entrada = new Scanner( System.in );

      int x; // primer número introducido por el usuario
      int y; // segundo número introducido por el usuario
```

```
        int z; // tercer número introducido por el usuario
        int resultado; // producto de los números

        System.out.print( "Escriba el primer entero: " ); // indicador de entrada
        x = entrada.nextInt(); // lee el primer entero

        System.out.print( "Escriba el segundo entero: " ); // indicador de entrada
        y = entrada.nextInt(); // lee el segundo entero

        System.out.print( "Escriba el tercer entero: " ); // indicador de entrada
        z = entrada.nextInt(); // lee el tercer entero

        resultado = x * y * z; // calcula el producto de los números

        System.out.printf( "El producto es %d\n", resultado );

    } // fin del método main

} // fin de la clase Producto
```

```
Escriba el primer entero: 10
Escriba el segundo entero: 20
Escriba el tercer entero: 30
El producto es 6000
```

Ejercicios

2.7 Complete las siguientes oraciones:

a) ______________ se utilizan para documentar un programa y mejorar su legibilidad.

b) Una decisión puede tomarse en un programa en Java con un(a) ______________.

c) Los cálculos se realizan normalmente mediante instrucciones ______________.

d) Los operadores aritméticos con la misma precedencia que la multiplicación son ______________ y ______________.

e) Cuando los paréntesis en una expresión aritmética están anidados, el conjunto ______________ de paréntesis se evalúa primero.

f) Una ubicación en la memoria de la computadora que puede contener distintos valores en diversos instantes de tiempo, durante la ejecución de un programa, se llama ______________.

2.8 Escriba instrucciones en Java que realicen cada una de las siguientes tareas:

a) Mostrar el mensaje `"Escriba un entero:"`, dejando el cursor en la misma línea.

b) Asignar el producto de las variables `b` y `c` a la variable `a`.

c) Indicar que un programa va a realizar un cálculo de nómina de muestra (es decir, usar texto que ayude a documentar un programa).

2.9 Conteste con *verdadero* o *falso* a cada una de las siguientes proposiciones; en caso de ser *falso*, explique por qué.

a) Los operadores en Java se evalúan de izquierda a derecha.

b) Los siguientes nombres de variables son todos válidos: `_barra_inferior_`, `m928134`, `t5`, `j7`, `sus_ventas$`, `su_$cuenta_total`, `a`, `b$`, `c`, `z` y `z2`.

c) Una expresión aritmética válida en Java sin paréntesis se evalúa de izquierda a derecha.

d) Los siguientes nombres de variables son todos inválidos: `3g`, `87`, `67h2`, `h22` y `2h`.

2.10 Suponiendo que `x = 2` y `y = 3`, ¿qué muestra cada una de las siguientes instrucciones?

a) `System.out.printf( "x = %d\n", x );`

b) `System.out.printf( "El valor de %d + %d es %d\n", x, x, ( x + x ) );`

c) `System.out.printf( "x =" );`

d) `System.out.printf( "%d = %d\n", ( x + y ), ( y + x ) );`

2.11 ¿Cuáles de las siguientes instrucciones de Java contienen variables, cuyos valores se modifican?

a) `p = i + j + k + 7;`
b) `System.out.println( "variables cuyos valores se destruyen" );`
c) `System.out.println( "a = 5" );`
d) `valor = entrada.nextInt();`

2.12 Dado que $y = ax^3 + 7$, ¿cuáles de las siguientes instrucciones en Java son correctas para esta ecuación?

a) `y = a * x * x * x + 7;`
b) `y = a * x * x * ( x + 7 );`
c) `y = ( a * x ) * x * ( x + 7 );`
d) `y = ( a * x ) * x * x + 7;`
e) `y = a * ( x * x * x ) + 7;`
f) `y = a * x * ( x * x + 7 );`

2.13 Indique el orden de evaluación de los operadores en cada una de las siguientes instrucciones en Java, y muestre el valor `x` después de ejecutar cada una de ellas:

a) `x = 7 + 3 * 6 / 2 - 1;`
b) `x = 2 % 2 + 2 * 2 - 2 / 2;`
c) `x = ( 3 * 9 * ( 3 + ( 9 * 3 / ( 3 ) ) ) );`

2.14 Escriba una aplicación que muestre los números del 1 al 4 en la misma línea, con cada par de números adyacentes separado por un espacio. Escriba el programa utilizando las siguientes técnicas:

a) Utilizando una instrucción `System.out.println`.
b) Utilizando cuatro instrucciones `System.out.print`.
c) Utilizando una instrucción `System.out.printf`.

2.15 Escriba una aplicación que pida al usuario que escriba dos números, que obtenga los números del usuario e imprima la suma, producto, diferencia y cociente (división) de los números. Use las técnicas que se muestran en la figura 2.7.

2.16 Escriba una aplicación que pida al usuario que escriba dos enteros, que obtenga los números del usuario y muestre el número más grande, seguido de las palabras `"es más grande"`. Si los números son iguales, imprima el mensaje `"Estos números son iguales"`. Utilice las técnicas que se muestran en la figura 2.15.

2.17 Escriba una aplicación que reciba tres enteros del usuario y muestre la suma, promedio, producto, menor y mayor de esos números. Utilice las técnicas que se muestran en la figura 2.15. [*Nota:* el cálculo del promedio en este ejercicio debe resultar en una representación entera del promedio. Por lo tanto, si la suma de los valores es 7, el promedio debe ser 2, no 2.3333...].

2.18 Escriba una aplicación que muestre un cuadro, un óvalo, una flecha y un diamante usando asteriscos (*), como se muestra a continuación:

```
*********   ***     *        *
*       *  *   *   ***      * *
*       * *     * *****    *   *
*       * *     *   *     *     *
*       * *     *   *    *       *
*       * *     *   *     *     *
*       * *     *   *      *   *
*       *  *   *    *       * *
*********   ***     *        *
```

2.19 ¿Qué imprime el siguiente código?

```
System.out.println( "*\n**\n***\n****\n*****" );
```

2.20 ¿Qué imprime el siguiente código?

```
System.out.println( "*" );
System.out.println( "***" );
System.out.println( "*****" );
System.out.println( "****" );
System.out.println( "**" );
```

2.21 ¿Qué imprime el siguiente código?

```
System.out.print( "*" );
System.out.print( "***" );
System.out.print( "*****" );
System.out.print( "****" );
System.out.println( "**" );
```

2.22 ¿Qué imprime el siguiente código?

```
System.out.print( "*" );
System.out.println( "***" );
System.out.println( "*****" );
System.out.print( "****" );
System.out.println( "**" );
```

2.23 ¿Qué imprime el siguiente código?

```
System.out.printf( "%s\n%s\n%s\n", "*", "***", "*****" );
```

2.24 Escriba una aplicación que lea cinco enteros y que determine e imprima los enteros mayor y menor en el grupo. Use solamente las técnicas de programación que aprendió en este capítulo.

2.25 Escriba una aplicación que lea un entero y que determine e imprima si es impar o par. [*Sugerencia:* use el operador residuo. Un número par es un múltiplo de 2. Cualquier múltiplo de 2 deja un residuo de 0 cuando se divide entre 2].

2.26 Escriba una aplicación que lea dos enteros, determine si el primero es un múltiplo del segundo e imprima el resultado. [*Sugerencia:* use el operador residuo].

2.27 Escriba una aplicación que muestre un patrón de tablero de damas, como se muestra a continuación:

```
* * * * * * * *
 * * * * * * * *
* * * * * * * *
 * * * * * * * *
* * * * * * * *
 * * * * * * * *
* * * * * * * *
 * * * * * * * *
```

2.28 He aquí un adelanto. En este capítulo, aprendió sobre los enteros y el tipo `int`. Java también puede representar números de punto flotante que contienen puntos decimales, como 3.14159. Escriba una aplicación que reciba del usuario el radio de un círculo como un entero, y que imprima el diámetro, la circunferencia y el área del círculo mediante el uso del valor de punto flotante 3.14159 para π. Use las técnicas que se muestran en la figura 2.7. [*Nota:* también puede utilizar la constante predefinida `Math.PI` para el valor de π. Esta constante es más precisa que el valor 3.14159. La clase `Math` se define en el paquete `java.lang`. Las clases en este paquete se importan de manera automática, por lo que no necesita importar la clase `Math` mediante la instrucción `import` para usarla]. Use las siguientes fórmulas (*r* es el radio):

$$\textit{diámetro} = 2r$$
$$\textit{circunferencia} = 2\pi r$$
$$\textit{área} = \pi r^2$$

No almacene los resultados de cada cálculo en una variable. En vez de ello, especifique cada cálculo como el valor que se imprimirá en una instrucción `System.out.printf`. Observe que los valores producidos por los cálculos del área y la circunferencia son números de punto flotante. Dichos valores pueden imprimirse con el especificador de formato `%f` en una instrucción `System.out.printf`. En el capítulo 3 aprenderá más acerca de los números de punto flotante.

2.29 He aquí otro adelanto. En este capítulo, aprendió acerca de los enteros y el tipo `int`. Java puede también representar letras en mayúsculas, en minúsculas y una considerable variedad de símbolos especiales. Cada carácter tiene su correspondiente representación entera. El conjunto de caracteres que utiliza una computadora, y las correspondientes representaciones enteras de esos caracteres, se conocen como el conjunto de caracteres de esa computadora. Usted puede indicar un valor de carácter en un programa con sólo encerrar ese carácter entre comillas sencillas, como en `'A'`.

Usted puede determinar el equivalente entero de un carácter si antepone a ese carácter la palabra (`int`), como en

```
(int) 'A'
```

Esta forma se conoce como operador de conversión de tipo. (Hablaremos más sobre estos operadores en el capítulo 4). La siguiente instrucción imprime un carácter y su equivalente entero:

```
System.out.printf(
   "El caracter %c tiene el valor %d\n", 'A', ( (int) 'A' ) );
```

Cuando se ejecuta esta instrucción, muestra el carácter `A` y el valor 65 (del conjunto de caracteres conocido como Unicode®) como parte de la cadena. Observe que el especificador de formato `%c` es un receptáculo para un carácter (en este caso, el carácter `'A'`).

Utilizando instrucciones similares a la mostrada anteriormente en este ejercicio, escriba una aplicación que muestre los equivalentes enteros de algunas letras en mayúsculas, en minúsculas, dígitos y símbolos especiales. Muestre los equivalentes enteros de los siguientes caracteres: `A B C a b c 0 1 2 $ * + /` y el carácter en blanco.

2.30 Escriba una aplicación que reciba del usuario un número compuesto por cinco dígitos, que separe ese número en sus dígitos individuales y los imprima, cada uno separado de los demás por tres espacios. Por ejemplo, si el usuario escribe el número `42339`, el programa debe imprimir

```
4   2   3   3   9
```

Suponga que el usuario escribe el número correcto de dígitos. ¿Qué ocurre cuando ejecuta el programa y escribe un número con más de cinco dígitos? ¿Qué ocurre cuando ejecuta el programa y escribe un número con menos de cinco dígitos? [*Sugerencia:* es posible hacer este ejercicio con las técnicas que aprendió en este capítulo. Necesitará utilizar los operadores de división y residuo para "seleccionar" cada dígito].

2.31 Utilizando sólo las técnicas de programación que aprendió en este capítulo, escriba una aplicación que calcule los cuadrados y cubos de los números del 0 al 10, y que imprima los valores resultantes en formato de tabla, como se muestra a continuación. [*Nota:* Este programa no requiere de ningún tipo de entrada por parte del usuario].

```
numero  cuadrado  cubo
0       0         0
1       1         1
2       4         8
3       9         27
4       16        64
5       25        125
6       36        216
7       49        343
8       64        512
9       81        729
10      100       1000
```

2.32 Escriba un programa que reciba cinco números, y que determine e imprima la cantidad de números negativos, positivos, y la cantidad de ceros recibidos.

3

Introducción a las clases y los objetos



Usted verá algo nuevo.
Dos cosas. Y las llamo
Cosa Uno y Cosa Dos.

—Dr. Theodor Seuss Geisel.

Nada puede tener valor
sin ser un objeto de utilidad.

—Karl Marx

Sus sirvientes públicos le
sirven bien.

—Adlai E. Stevenson

Saber cómo responder
a alguien que habla,
contestar a alguien que
envía un mensaje.

—Amenemope

OBJETIVOS

En este capítulo aprenderá a:

- Comprender qué son las clases, los objetos, los métodos y las variables de instancia.
- Declarar una clase y utilizarla para crear un objeto.
- Declarar métodos en una clase para implementar los comportamientos de ésta.
- Declarar variables de instancia en una clase para implementar los atributos de ésta.
- Saber cómo llamar a los métodos de un objeto para hacer que realicen sus tareas.
- Conocer las diferencias entre las variables de instancia de una clase y las variables locales de un método.
- Utilizar un constructor para asegurar que los datos de un objeto se inicialicen cuando se cree el objeto.
- Conocer las diferencias entre los tipos primitivos y los tipos por referencia.

Plan general

3.1 Introducción

En la sección 1.16 le presentamos la terminología básica y los conceptos acerca de la programación orientada a objetos. El capítulo 2 comenzó a utilizar esos conceptos para crear aplicaciones simples que mostraran mensajes al usuario, que obtuvieran información del usuario, realizaran cálculos y tomaran decisiones. Una característica común de todas las aplicaciones del capítulo 2 fue que todas las instrucciones que realizaban tareas se encontraban en el método `main`. Por lo general, las aplicaciones que usted desarrollará en este libro consistirán de dos o más clases, cada una de las cuales contendrá dos o más métodos. Si usted se integra a un equipo de desarrollo en la industria, podría trabajar en aplicaciones que contengan cientos, o incluso hasta miles de clases. En este capítulo presentaremos un marco de trabajo simple para organizar las aplicaciones orientadas a objetos en Java.

Primero explicaremos el concepto de las clases mediante el uso de un ejemplo real. Después presentaremos cinco aplicaciones completas para demostrarle cómo crear y utilizar sus propias clases. Los primeros cuatro ejemplos empiezan nuestro ejemplo práctico acerca de cómo desarrollar una clase tipo libro de calificaciones, que los instructores pueden utilizar para mantener las calificaciones de las pruebas de sus estudiantes. Durante los siguientes capítulos ampliaremos este ejemplo práctico y culminaremos con la versión que se presenta en el capítulo 7, Arreglos. El último ejemplo en este capítulo introduce los números de punto flotante (es decir, números que contienen puntos decimales, como 0.0345, –7.23 y 100.7) en el contexto de una clase tipo cuenta bancaria, la cual mantiene el saldo de un cliente.

3.2 Clases, objetos, métodos y variables de instancia

Comenzaremos con una analogía simple, para ayudarle a comprender el concepto de las clases y su contenido. Suponga que desea conducir un automóvil y, para hacer que aumente su velocidad, debe presionar el pedal del acelerador. ¿Qué debe ocurrir antes de que pueda hacer esto? Bueno, antes de poder conducir un automóvil, alguien tiene que diseñarlo. Por lo general, un automóvil empieza en forma de dibujos de ingeniería, similares a los planos de construcción que se utilizan para diseñar una casa. Estos dibujos de ingeniería incluyen el diseño del pedal del acelerador, para que el automóvil aumente su velocidad. El pedal "oculta" los complejos mecanismos que se encargan de que el automóvil aumente su velocidad, de igual forma que el pedal del freno "oculta" los mecanismos que disminuyen la velocidad del automóvil y por otro lado, el volante "oculta" los mecanismos que hacen que el automóvil de vuelta. Esto permite que las personas con poco o nada de conocimiento acerca de cómo funcionan los motores puedan conducir un automóvil con facilidad.

Desafortunadamente, no puede conducir los dibujos de ingeniería de un auto. Antes de poder conducir un automóvil, éste debe construirse a partir de los dibujos de ingeniería que lo describen. Un automóvil completo tendrá un pedal acelerador verdadero para hacer que aumente su velocidad, pero aún así no es suficiente; el automóvil no acelerará por su propia cuenta, así que el conductor debe oprimir el pedal del acelerador.

Ahora utilizaremos nuestro ejemplo del automóvil para introducir los conceptos clave de programación de esta sección. Para realizar una tarea en una aplicación se requiere un método. El método describe los mecanismos

que se encargan de realizar sus tareas; y oculta al usuario las tareas complejas que realiza, de la misma forma que el pedal del acelerador de un automóvil oculta al conductor los complejos mecanismos para hacer que el automóvil vaya más rápido. En Java, empezamos por crear una unidad de aplicación llamada clase para alojar a un método, así como los dibujos de ingeniería de un automóvil alojan el diseño del pedal del acelerador. En una clase se proporcionan uno o más métodos, los cuales están diseñados para realizar las tareas de esa clase. Por ejemplo, una clase que representa a una cuenta bancaria podría contener un método para depositar dinero en una cuenta, otro para retirar dinero de una cuenta y un tercero para solicitar el saldo actual de la cuenta.

Así como no podemos conducir un dibujo de ingeniería de un automóvil, tampoco podemos "conducir" una clase. De la misma forma que alguien tiene que construir un automóvil a partir de sus dibujos de ingeniería para poder conducirlo, también debemos construir un objeto de una clase para poder hacer que un programa realice las tareas que la clase le describe cómo realizar. Ésta es una de las razones por las cuales Java se conoce como un lenguaje de programación orientado a objetos.

Cuando usted conduce un automóvil, si oprime el pedal del acelerador se envía un mensaje al automóvil para que realice una tarea-hacer que el automóvil vaya más rápido. De manera similar, se envían **mensajes** a un objeto; cada mensaje se conoce como la **llamada a un método**, e indica a un método del objeto que realice su tarea.

Hasta ahora, hemos utilizado la analogía del automóvil para introducir las clases, los objetos y los métodos. Además de las capacidades con las que cuenta un automóvil, también tiene muchos atributos como su color, el número de puertas, la cantidad de gasolina en su tanque, su velocidad actual y el total de kilómetros recorridos (es decir, la lectura de su odómetro). Al igual que las capacidades del automóvil, estos atributos se representan como parte del diseño en sus diagramas de ingeniería. Cuando usted conduce un automóvil, estos atributos siempre están asociados con él. Cada uno mantiene sus propios atributos. Por ejemplo, cada conductor sabe cuánta gasolina tiene en su propio tanque, pero no cuánta hay en los tanques de otros automóviles. De manera similar, un objeto tiene atributos que lleva consigo cuando se utiliza en un programa. Éstos se especifican como parte de la clase del objeto. Por ejemplo, un objeto tipo cuenta bancaria tiene un atributo llamado saldo, el cual representa la cantidad de dinero en la cuenta. Cada objeto tipo cuenta bancaria conoce el saldo en la cuenta que representa, pero no los saldos de las otras cuentas en el banco. Los atributos se especifican mediante las **variables de instancia** de la clase.

El resto de este capítulo presenta ejemplos que demuestran los conceptos que presentamos aquí, dentro del contexto de la analogía del automóvil. Los primeros cuatro ejemplos se encargan de construir en forma incremental una clase llamada `LibroCalificaciones` para demostrar estos conceptos:

1. El primer ejemplo presenta una clase llamada `LibroCalificaciones`, con un método que simplemente muestra un mensaje de bienvenida cuando se le llama. Después le mostraremos cómo crear un objeto de esa clase y cómo llamarlo, para que muestre el mensaje de bienvenida.
2. El segundo ejemplo modifica el primero, al permitir que el método reciba el nombre de un curso como argumento, y al mostrar ese nombre como parte del mensaje de bienvenida.
3. El tercer ejemplo muestra cómo almacenar el nombre del curso en un objeto tipo `LibroCalificaciones`. Para esta versión de la clase, también le mostraremos cómo utilizar los métodos para establecer el nombre del curso y obtener este nombre.
4. El cuarto ejemplo demuestra cómo pueden inicializarse los datos en un objeto tipo `LibroCalificaciones`, a la hora de crear el objeto; el constructor de la clase se encarga de realizar el proceso de inicialización.

El último ejemplo en el capítulo presenta una clase llamada `Cuenta`, la cual refuerza los conceptos presentados en los primeros cuatro ejemplos, e introduce los números de punto flotante. Para este fin, presentamos una clase llamada `Cuenta`, la cual representa una cuenta bancaria y mantiene su saldo como un número de punto flotante. La clase contiene dos métodos —uno para acreditar un depósito a la cuenta, con lo cual se incrementa el saldo, y otro para obtener el saldo. El constructor de la clase permite inicializar el saldo de cada objeto tipo `Cuenta`, a la hora de crear el objeto. Crearemos dos objetos tipo `Cuenta` y haremos depósitos en cada uno de ellos, para mostrar que cada objeto mantiene su propio saldo. El ejemplo también demuestra cómo recibir e imprimir en pantalla números de punto flotante.

3.3 Declaración de una clase con un método e instanciamiento de un objeto de una clase

Comenzaremos con un ejemplo que consiste en las clases `LibroCalificaciones` (figura 3.1) y `PruebaLibroCalificaciones` (figura 3.2). La clase `LibroCalificaciones` (declarada en el archivo `LibroCalificaciones.java`)

se utilizará para mostrar un mensaje en la pantalla (figura 3.2), para dar la bienvenida al instructor a la aplicación del libro de calificaciones. La clase `PruebaLibroCalificaciones` (declarada en el archivo `PruebaLibroCalificaciones.java`) es una clase de aplicación en la que el método `main` utilizará a la clase `LibroCalificaciones`. Cada declaración de clase que comienza con la palabra clave `public` debe almacenarse en un archivo que tenga el mismo nombre que la clase, y que termine con la extensión de archivo `.java`. Por lo tanto, las clases `LibroCalificaciones` y `PruebaLibroCalificaciones` deben declararse en archivos separados, ya que cada clase se declara como `public`.



Error común de programación 3.1

Declarar más de una clase `public` en el mismo archivo es un error de compilación.

La clase `LibroCalificaciones`

La declaración de la clase `LibroCalificaciones` (figura 3.1) contiene un método llamado `mostrarMensaje` (líneas 7-10), el cual muestra un mensaje en la pantalla. La línea 9 de la clase realiza el trabajo de mostrar el mensaje. Recuerde que una clase es como un plano de construcción; necesitamos crear un objeto de esta clase y llamar a su método para hacer que se ejecute la línea 9 y que muestre su mensaje.

La declaración de la clase empieza en la línea 4. La palabra clave `public` es un **modificador de acceso**. Por ahora, simplemente declararemos cada clase como `public`. Toda declaración de clase contiene la palabra clave `class`, seguida inmediatamente por el nombre de la clase. El cuerpo de toda clase se encierra entre una llave izquierda y una derecha ({ y }), como en las líneas 5 y 12 de la clase `LibroCalificaciones`.

En el capítulo 2, cada clase que declaramos tenía un método llamado `main`. La clase `LibroCalificaciones` también tiene un método: `mostrarMensaje` (líneas 7-10). Recuerde que `main` es un método especial, que siempre es llamado, automáticamente, por la Máquina Virtual de Java (JVM) a la hora de ejecutar una aplicación. La mayoría de los métodos no se llaman en forma automática. Como veremos en breve, es necesario llamar al método `mostrarMensaje` para decirle que haga su trabajo.

La declaración del método comienza con la palabra clave `public` para indicar que el método está "disponible al público"; es decir, los métodos de otras clases pueden llamarlo desde el exterior del cuerpo de la declaración de la clase. La palabra clave `void` indica que este método realizará una tarea pero no devolverá (es decir, regresará) información al **método que hizo la llamada** cuando complete su tarea. Ya hemos utilizado métodos que devuelven información; por ejemplo, en el capítulo 2 utilizó el método `nextInt` de `Scanner` para recibir un entero escrito por el usuario desde el teclado. Cuando `nextInt` recibe un valor de entrada, devuelve ese valor para utilizarlo en el programa.

El nombre del método, `mostrarMensaje`, va después del tipo de valor de retorno. Por convención, los nombres de los métodos comienzan con una letra minúscula, y el resto de las palabras en el nombre empiezan con letra mayúscula. Los paréntesis después del nombre del método indican que éste es un método. Un conjunto vacío de paréntesis, como se muestra en la línea 7, indica que este método no requiere información adicional para realizar su tarea. La línea 7 se conoce comúnmente como el **encabezado del método**. El cuerpo de cada método se delimita mediante una llave izquierda y una llave derecha ({ y }), como en las líneas 8 y 10.

```
1  // Fig. 3.1: LibroCalificaciones.java
2  // Declaración de una clase con un método.
3
4  public class LibroCalificaciones
5  {
6     // muestra un mensaje de bienvenida al usuario de LibroCalificaciones
7     public void mostrarMensaje()
8     {
9        System.out.println( “Bienvenido al Libro de calificaciones!” );
10    } // fin del método mostrarMensaje
11
12 } // fin de la clase LibroCalificaciones
```

Figura 3.1 | Declaración de una clase con un método.

El cuerpo de un método contiene una o varias instrucciones que realizan su trabajo. En este caso, el método contiene una instrucción (línea 9) que muestra el mensaje `"Bienvenido al Libro de calificaciones!"`, seguido de una nueva línea en la ventana de comandos. Una vez que se ejecuta esta instrucción, el método ha completado su trabajo.

A continuación, nos gustaría utilizar la clase `LibroCalificaciones` en una aplicación. Como aprendió en el capítulo 2, el método `main` empieza la ejecución de todas las aplicaciones. Una clase que contiene el método `main` es una aplicación de Java. Dicha clase es especial, ya que la JVM puede utilizar a `main` como un punto de entrada para empezar la ejecución. La clase `LibroCalificaciones` no es una aplicación, ya que no contiene a `main`. Por lo tanto, si trata de ejecutar `LibroCalificaciones` escribiendo `java LibroCalificaciones` en la ventana de comandos, recibirá un mensaje de error como este:

```
Exception in thread "main" java.lang.NoSuchMethodError: main
```

Esto no fue un problema en el capítulo 2, ya que cada clase que declaramos tenía un método `main`. Para corregir este problema con la clase `LibroCalificaciones`, debemos declarar una clase separada que contenga un método `main`, o colocar un método `main` en la clase `LibroCalificaciones`. Para ayudarlo a prepararse para los programas más extensos que encontrará más adelante en este libro y en la industria, utilizamos una clase separada (`PruebaLibroCalificaciones` en este ejemplo) que contiene el método `main` para probar cada nueva clase que vayamos a crear en este capítulo.

La clase `PruebaLibroCalificaciones`

La declaración de la clase `PruebaLibroCalificaciones` (figura 3.2) contiene el método `main` que controlará la ejecución de nuestra aplicación. Cualquier clase que contiene el método `main`, declarado como se muestra en la línea 7, puede utilizarse para ejecutar una aplicación. La declaración de la clase `PruebaLibroCalificaciones` empieza en la línea 4 y termina en la línea 16. La clase sólo contiene un método `main`, algo común en muchas clases que empiezan la ejecución de una aplicación.

Las líneas 7 a la 14 declaran el método `main`. En el capítulo 2 vimos que el encabezado `main` debe aparecer como se muestra en la línea 7; en caso contrario, no se ejecutará la aplicación. Una parte clave para permitir que la JVM localice y llame al método `main` para empezar la ejecución de la aplicación es la palabra clave `static` (línea 7), la cual indica que `main` es un método `static`. Un método `static` es especial, ya que puede llamarse sin tener que crear primero un objeto de la clase en la cual se declara ese método. En el capítulo 6, Métodos: un análisis más detallado, explicaremos a detalle los métodos `static`.

```
1  // Fig. 3.2: PruebaLibroCalificaciones.java
2  // Crea un objeto LibroCalificaciones y llama a su método mostrarMensaje.
3
4  public class PruebaLibroCalificaciones
5  {
6     // el método main empieza la ejecución del programa
7     public static void main( String args[] )
8     {
9        // crea un objeto LibroCalificaciones y lo asigna a miLibroCalificaciones
10       LibroCalificaciones miLibroCalificaciones = new LibroCalificaciones();
11
12       // llama al método mostrarMensaje de miLibroCalificaciones
13       miLibroCalificaciones.mostrarMensaje();
14    } // fin de main
15
16 } // fin de la clase PruebaLibroCalificaciones
```

```
Bienvenido al Libro Calificaciones!
```

Figura 3.2 | Cómo crear un objeto de la clase `LibroCalificaciones` y llamar a su método `mostrarMensaje`.

En esta aplicación nos gustaría llamar al método `mostrarMensaje` de la clase `LibroCalificaciones` para mostrar el mensaje de bienvenida en la ventana de comandos. Por lo general, no podemos llamar a un método que pertenece a otra clase, sino hasta crear un objeto de esa clase, como se muestra en la línea 10. Empezaremos por declarar la variable `miLibroCalificaciones`. Observe que el tipo de la variable es `LibroCalificaciones`; la clase que declaramos en la figura 3.1. Cada nueva clase que creamos se convierte en un nuevo tipo, que puede usarse para declarar variables y crear objetos. Los programadores pueden declarar nuevos tipos de clases según lo necesiten; ésta es una razón por la cual Java se conoce como un **lenguaje extensible**.

La variable `miLibroCalificaciones` se inicializa con el resultado de la **expresión de creación de instancia de clase** `new LibroCalificaciones()`. La palabra clave `new` crea un nuevo objeto de la clase especificada a la derecha de la palabra clave (es decir, `LibroCalificaciones`). Los paréntesis a la derecha de `LibroCalificaciones` son obligatorios. Como veremos en la sección 3.7, esos paréntesis en combinación con el nombre de una clase representan una llamada a un **constructor**, que es similar a un método, pero se utiliza sólo cuando se crea un objeto, para inicializar los datos de éste. En esa sección verá que los datos pueden colocarse entre paréntesis para especificar los valores iniciales para los datos del objeto. Por ahora, sólo dejaremos los paréntesis vacíos.

Así como podemos usar el objeto `System.out` para llamar a los métodos `print`, `printf` y `println`, también podemos usar el objeto `miLibroCalificaciones` para llamar al método `mostrarMensaje`. La línea 13 llama al método `mostrarMensaje` (líneas 7-10 de la figura 3.1), usando `miLibroCalificaciones` seguida de un **separador punto** (`.`), el nombre del método `mostrarMensaje` y un conjunto vacío de paréntesis. Esta llamada hace que el método `mostrarMensaje` realice su tarea. La llamada a este método difiere de las del capítulo 2 en las que se mostraba la información en una ventana de comandos; cada una de estas llamadas al método proporcionaban argumentos que especificaban los datos a mostrar. Al inicio de la línea 13, "`miLibroCalificaciones`". Indica que `main` debe utilizar el objeto `miLibroCalificaciones` que se creó en la línea 10. La línea 7 de la figura 3.1 indica que el método `mostrarMensaje` tiene una lista de parámetros vacía; es decir, `mostrarMensaje` no requiere información adicional para realizar su tarea. Por esta razón, la llamada al método (línea 13 de la figura 3.2) especifica un conjunto vacío de paréntesis después del nombre del método, para indicar que no se van a pasar argumentos al método `mostrarMensaje`. Cuando el método `mostrarMensaje` completa su tarea, el método `main` continúa su ejecución en la línea 14. Éste es el final del método `main`, por lo que el programa termina.

Compilación de una aplicación con varias clases

Debe compilar las clases de las figuras 3.1 y 3.2 antes de poder ejecutar la aplicación. Primero, cambie al directorio que contiene los archivos de código fuente de la aplicación. Después, escriba el comando

```
javac LibroCalificaciones.java PruebaLibroCalificaciones.java
```

para compilar ambas clases a la vez. Si el directorio que contiene la aplicación sólo incluye los archivos de esta aplicación, puede compilar todas las clases que haya en el directorio con el comando

```
javac *.java
```

El asterisco (`*`) en `*.java` indica que deben compilarse todos los archivos en el directorio actual que terminen con la extensión de nombre de archivo "`.java`".

Diagrama de clases de UML para la clase `LibroCalificaciones`

La figura 3.3 presenta un **diagrama de clases de UML** para la clase `LibroCalificaciones` de la figura 3.1. En la sección 1.16 vimos que UML es un lenguaje gráfico, utilizado por los programadores para representar sistemas orientados a objetos en forma estandarizada. En UML, cada clase se modela en un diagrama de clases en forma de un rectángulo con tres componentes. El compartimiento superior contiene el nombre de la clase, centrado en forma horizontal y en negrita. El compartimiento de en medio contiene los atributos de la clase, que en Java corresponden a las variables de instancia. En la figura 3.3, el compartimiento de en medio está vacío, ya que la versión de la clase `LibroCalificaciones` en la figura 3.1 no tiene atributos. El compartimiento inferior contiene las operaciones de la clase, que en Java corresponden a los métodos. Para modelar las operaciones, UML lista el nombre de la operación precedido por un modificador de acceso y seguido de un conjunto de paréntesis. La clase `LibroCalificaciones` tiene un solo método llamado `mostrarMensaje`, por lo que el compartimiento inferior de la figura 3.3 lista una operación con este nombre. El método `mostrarMensaje` no requiere información adicional para realizar sus tareas, por lo que los paréntesis que van después del nombre del método en el diagrama de



Figura 3.3 | Diagrama de clases de UML, el cual indica que la clase `LibroCalificaciones` tiene una operación `public` llamada `mostrarMensaje`.

clases están vacíos, de igual forma que como aparecieron en la declaración del método, en la línea 7 de la figura 3.1. El signo más (+) que va antes del nombre de la operación indica que `mostrarMensaje` es una operación `public` en UML (es decir, un método `public` en Java). Utilizaremos los diagramas de clases de UML a menudo para sintetizar los atributos y las operaciones de una clase.

3.4 Declaración de un método con un parámetro

En nuestra analogía del automóvil de la sección 3.2, hablamos sobre el hecho de que al oprimir el pedal del acelerador se envía un mensaje al automóvil para que realice una tarea: hacer que vaya más rápido. Pero ¿qué tan rápido debería acelerar el automóvil? Como sabe, entre más oprima el pedal, mayor será la aceleración del automóvil. Por lo tanto, el mensaje para el automóvil en realidad incluye tanto la tarea a realizar como información adicional que ayuda al automóvil a ejecutar su tarea. A la información adicional se le conoce como **parámetro**; el valor del parámetro ayuda al automóvil a determinar qué tan rápido debe acelerar. De manera similar, un método puede requerir uno o más parámetros que representan la información adicional que necesita para realizar su tarea. La llamada a un método proporciona valores (llamados argumentos) para cada uno de los parámetros de ese método. Por ejemplo, el método `System.out.println` requiere un argumento que especifica los datos a mostrar en una ventana de comandos. De manera similar, para realizar un depósito en una cuenta bancaria, un método llamado `deposito` especifica un parámetro que representa el monto a depositar. Cuando se hace una llamada al método `deposito`, se asigna al parámetro del método un valor como argumento, que representa el monto a depositar. Entonces, el método realiza un depósito por ese monto.

Nuestro siguiente ejemplo declara la clase `LibroCalificaciones` (figura 3.4), con un método `mostrarMensaje` que muestra el nombre del curso como parte del mensaje de bienvenida (en la figura 3.5 podrá ver la ejecución de ejemplo). El nuevo método `mostrarMensaje` requiere un parámetro que representa el nombre del curso a imprimir en pantalla.

Antes de hablar sobre las nuevas características de la clase `LibroCalificaciones`, veamos cómo se utiliza la nueva clase desde el método `main` de la clase `PruebaLibroCalificaciones` (figura 3.5). La línea 12 crea un objeto `Scanner` llamado `entrada`, para recibir el nombre del curso escrito por el usuario. La línea 15 crea un objeto de la clase `LibroCalificaciones` y lo asigna a la variable `miLibroCalificaciones`. La línea 18 pide al usuario que escriba el nombre de un curso. La línea 19 lee el nombre que introduce el usuario y lo asigna a la variable `nombreDelCurso`, mediante el uso del método `nextLine` de `Scanner` para realizar la operación de entrada. El

```java
// Fig. 3.4: LibroCalificaciones.java
// Declaración de una clase con un método que tiene un parámetro.

public class LibroCalificaciones
{
   // muestra un mensaje de bienvenida al usuario de LibroCalificaciones
   public void mostrarMensaje( String nombreDelCurso )
   {
      System.out.printf( "Bienvenido al libro de calificaciones para\n%s!\n",
         nombreDelCurso );
   } // fin del método mostrarMensaje

} // fin de la clase LibroCalificaciones
```

Figura 3.4 | Declaración de una clase con un método que tiene un parámetro.

```
1  // Fig. 3.5: PruebaLibroCalificaciones.java
2  // Crea un objeto LibroCalificaciones y pasa un objeto String
3  // a su método mostrarMensaje.
4  import java.util.Scanner; // el programa usa la clase Scanner
5
6  public class PruebaLibroCalificaciones
7  {
8     // el método main empieza la ejecución del programa
9     public static void main( String args[] )
10    {
11       // crea un objeto Scanner para obtener la entrada de la ventana de comandos
12       Scanner entrada = new Scanner( System.in );
13
14       // crea un objeto LibroCalificaciones y lo asigna a miLibroCalificaciones
15       LibroCalificaciones miLibroCalificaciones = new LibroCalificaciones();
16
17       // pide y recibe el nombre del curso como entrada
18       System.out.println( "Escriba el nombre del curso:" );
19       String nombreDelCurso = entrada.nextLine(); // lee una línea de texto
20       System.out.println(); // imprime una línea en blanco
21
22       // llama al método mostrarMensaje de miLibroCalificaciones
23       // y pasa nombreDelCurso como argumento
24       miLibroCalificaciones.mostrarMensaje( nombreDelCurso );
25    } // fin de main
26
27 } // fin de la clase PruebaLibroCalificaciones
```

```
Escriba el nombre del curso:
CS101 Introduccion a la programacion en Java

Bienvenido al libro de calificaciones para
CS101 Introduccion a la programacion en Java!
```

Figura 3.5 | Cómo crear un objeto `LibroCalificaciones` y pasar un objeto `String` a su método `mostrarMensaje`.

usuario escribe el nombre del curso y oprime *Intro* para enviarlo al programa. Observe que al oprimir *Intro* se inserta un carácter de nueva línea al final de los caracteres escritos por el usuario. El método `nextLine` lee los caracteres que escribió el usuario hasta encontrar el carácter de nueva línea, y después devuelve un objeto `String` que contiene los caracteres hasta, pero sin incluir, la nueva línea. El carácter de nueva línea se descarta. La clase `Scanner` también cuenta con un método similar (`next`) para leer palabras individuales. Cuando el usuario oprime *Intro* después de escribir la entrada, el método `next` lee caracteres hasta encontrar un carácter de espacio en blanco (espacio, tabulador o nueva línea), y después devuelve un objeto `String` que contiene los caracteres hasta, pero sin incluir, el carácter de espacio en blanco (que se descarta). No se pierde toda la información que va después del primer carácter de espacio en blanco; estará disponible para que la lean otras instrucciones que llamen a los métodos de `Scanner`, más adelante en el programa.

La línea 24 llama al método `mostrarMensaje` de `miLibroCalificaciones`. La variable `nombreDelCurso` entre paréntesis es el argumento que se pasa al método `mostrarMensaje`, para que éste pueda realizar su tarea. El valor de la variable `nombreDelCurso` en `main` se convierte en el valor del parámetro `nombreDelCurso` del método `mostrarMensaje`, en la línea 7 de la figura 3.4. Al ejecutar esta aplicación, observe que el método `mostrarMensaje` imprime en pantalla el nombre que usted escribió como parte del mensaje de bienvenida (figura 3.5).



Observación de ingeniería de software 3.1

Por lo general, los objetos se crean mediante el uso de `new`. *Una excepción es la literal de cadena que está encerrada entre comillas, como* "`hola`". *Las literales de cadena son referencias a objetos* `String` *que Java crea de manera implícita.*

Más sobre los argumentos y los parámetros

Al declarar un método, debe especificar si el método requiere datos para realizar su tarea. Para ello es necesario colocar información adicional en la **lista de parámetros** del método, la cual se encuentra en los paréntesis que van después del nombre del método. La lista de parámetros puede contener cualquier número de parámetros, incluso ninguno. Los paréntesis vacíos después del nombre del método (como en la figura 3.1, línea 7) indican que un método no requiere parámetros. En la figura 3.4, la lista de parámetros de `mostrarMensaje` (línea 7) declara que el método requiere un parámetro. Cada parámetro debe especificar un tipo y un identificador. En este caso, el tipo `String` y el identificador `nombreDelCurso` indican que el método `mostrarMensaje` requiere un objeto `String` para realizar su tarea. En el instante en que se llama al método, el valor del argumento en la llamada se asigna al parámetro correspondiente (en este caso, `nombreDelCurso`) en el encabezado del método. Después, el cuerpo del método utiliza el parámetro `nombreDelCurso` para acceder al valor. Las líneas 9 y 10 de la figura 3.4 muestran el valor del parámetro `nombreDelCurso`, mediante el uso del especificador de formato `%s` en la cadena de formato de `printf`. Observe que el nombre de la variable de parámetro (figura 3.4, línea 7) puede ser igual o distinto al nombre de la variable de argumento (figura 3.5, línea 24).

Un método puede especificar múltiples parámetros; sólo hay que separar un parámetro de otro mediante una coma (en el capítulo 6 veremos un ejemplo de esto). El número de argumentos en la llamada a un método debe coincidir con el número de parámetros en la lista de parámetros de la declaración del método que se llamó. Además, los tipos de los argumentos en la llamada al método deben ser "consistentes con" los tipos de los parámetros correspondientes en la declaración del método (como veremos en capítulos posteriores, no siempre se requiere que el tipo de un argumento y el tipo de su correspondiente parámetro sean idénticos). En nuestro ejemplo, la llamada al método pasa un argumento de tipo `String` (`nombreDelCurso` se declara como `String` en la línea 19 de la figura 3.5) y la declaración del método especifica un parámetro de tipo `String` (línea 7 en la figura 3.4). Por lo tanto, en este ejemplo, el tipo del argumento en la llamada al método coincide exactamente con el tipo del parámetro en el encabezado del método.



Error común de programación 3.2

Si el número de argumentos en la llamada a un método no coincide con el número de parámetros en la declaración del método, se produce un error de compilación.



Error común de programación 3.3

Si los tipos de los argumentos en la llamada a un método no son consistentes con los tipos de los parámetros correspondientes en la declaración del método, se produce un error de compilación.

Diagrama de clases de UML actualizado para la clase `LibroCalificaciones`

El diagrama de clases de UML de la figura 3.6 modela la clase `LibroCalificaciones` de la figura 3.4. Al igual que la Figura 3.1, esta clase `LibroCalificaciones` contiene la operación `public` llamada `mostrarMensaje`. Sin embargo, esta versión de `mostrarMensaje` tiene un parámetro. La forma en que UML modela un parámetro es un poco distinta a la de Java, ya que lista el nombre del parámetro, seguido de dos puntos y del tipo del parámetro entre paréntesis, después del nombre de la operación. UML tiene sus propios tipos de datos, que son similares a los de Java (pero como veremos, no todos los tipos de datos de UML tienen los mismos nombres que los tipos correspondientes en Java). El tipo `String` de UML corresponde al tipo `String` de Java. El método `mostrarMensaje` de `LibroCalificaciones` (figura 3.4) tiene un parámetro `String` llamado `nombreDelCurso`, por lo que en la figura 3.6 se lista a `nombreDelCurso : String` entre los paréntesis que van después de `mostrarMensaje`.

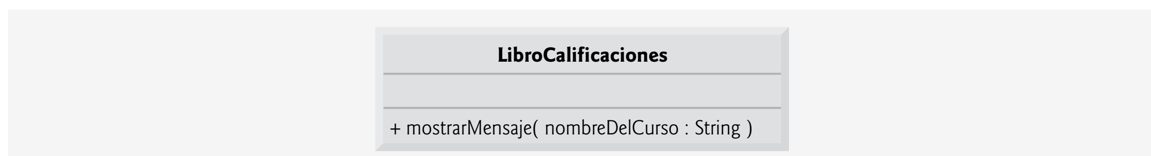


Figura 3.6 | Diagrama de clases de UML, que indica que la clase `LibroCalificaciones` tiene una operación llamada `mostrarMensaje`, con un parámetro llamado `nombreDelCurso` de tipo `String` de UML.

Observaciones acerca del uso de las declaraciones `import`

Observe la declaración `import` en la figura 3.5 (línea 4). Esto indica al compilador que el programa utiliza la clase `Scanner`. ¿Por qué necesitamos importar la clase `Scanner`, pero no las clases `System`, `String` o `LibroCalificaciones`? La mayoría de las clases que utilizará en los programas de Java deben importarse. Las clases `System` y `String` están en el paquete `java.lang`, que se importa de manera implícita en todo programa de Java, por lo que todos los programas pueden usar las clases del paquete `java.lang` sin tener que importarlas de manera explícita.

Hay una relación especial entre las clases que se compilan en el mismo directorio en el disco, como las clases `LibroCalificaciones` y `PruebaLibroCalificaciones`. De manera predeterminada, se considera que dichas clases se encuentran en el mismo paquete; a éste se le conoce como el **paquete predeterminado**. Las clases en el mismo paquete se importan implícitamente en los archivos de código fuente de las otras clases en el mismo paquete. Por ende, no se requiere una declaración `import` cuando la clase en un paquete utiliza a otra en el mismo paquete; como cuando `PruebaLibroCalificaciones` utiliza a la clase `LibroCalificaciones`.

La declaración `import` en la línea 4 no es obligatoria si siempre hacemos referencia a la clase `Scanner` como `java.util.Scanner`, que incluye el nombre completo del paquete y de la clase. Esto se conoce como el **nombre de clase completamente calificado**. Por ejemplo, la línea 12 podría escribirse como

```
java.util.Scanner entrada = new java.util.Scanner( System.in );
```



Observación de ingeniería de software 3.2

El compilador de Java no requiere declaraciones `import` *en un archivo de código fuente de Java, si se especifica el nombre de clase completamente calificado cada vez que se utilice el nombre de una clase en el código fuente. Pero la mayoría de los programadores de Java consideran que el uso de nombres completamente calificados es incómodo, por lo cual prefieren usar declaraciones* `import`.

3.5 Variables de instancia, métodos *establecer* y métodos *obtener*

En el capítulo 2 declaramos todas las variables de una aplicación en el método `main`. Las variables que se declaran en el cuerpo de un método específico se conocen como **variables locales**, y sólo se pueden utilizar en ese método. Cuando termina ese método, se pierden los valores de sus variables locales. En la sección 3.2 vimos que un objeto tiene atributos que lleva consigo cuando se utiliza en un programa. Dichos atributos existen antes de que un objeto llame a un método, y después de que el método completa su ejecución.

Por lo general, una clase consiste en uno o más métodos que manipulan los atributos pertenecientes a un objeto específico de la clase. Los atributos se representan como variables en la declaración de la clase. Dichas variables se llaman **campos**, y se declaran dentro de la declaración de una clase, pero fuera de los cuerpos de las declaraciones de los métodos de la clase. Cuando cada objeto de una clase mantiene su propia copia de un atributo, el campo que representa a ese atributo se conoce también como variable de instancia; cada objeto (instancia) de la clase tiene una instancia separada de la variable en memoria. El ejemplo en esta sección demuestra una clase `LibroCalificaciones`, que contiene una variable de instancia llamada `nombreDelCurso` para representar el nombre del curso de un objeto `LibroCalificaciones` específico.

La clase `LibroCalificaciones` *con una variable de instancia, un método* establecer *y un método* obtener

En nuestra siguiente aplicación (figuras 3.7 y 3.8), la clase `LibroCalificaciones` (figura 3.7) mantiene el nombre del curso como una variable de instancia, para que pueda usarse o modificarse en cualquier momento, durante la ejecución de una aplicación. Esta clase contiene tres métodos: `establecerNombreDelCurso`, `obtenerNombreDelCurso` y `mostrarMensaje`. El método `establecerNombreDelCurso` almacena el nombre de un curso en un `LibroCalificaciones`. El método `obtenerNombreDelCurso` obtiene el nombre del curso de un `LibroCalificaciones`. El método `mostrarMensaje`, que en este caso no especifica parámetros, sigue mostrando un mensaje de bienvenida que incluye el nombre del curso; como veremos más adelante, el método ahora obtiene el nombre del curso mediante una llamada a otro método en la misma clase: `obtenerNombreDelCurso`.

Por lo general, un instructor enseña más de un curso, cada uno con su propio nombre. La línea 7 declara que `nombreDelCurso` es una variable de tipo `String`. Como la variable se declara en el cuerpo de la clase, pero fuera de los cuerpos de los métodos de la misma (líneas 10 a la 13, 16 a la 19 y 22 a la 28), la línea 7 es una declaración para una variable de instancia. Cada instancia (es decir, objeto) de la clase `LibroCalificaciones` contiene una

```
1  // Fig. 3.7: LibroCalificaciones.java
2  // Clase LibroCalificaciones que contiene una variable de instancia nombreDelCurso
3  // y métodos para establecer y obtener su valor.
4
5  public class LibroCalificaciones
6  {
7     private String nombreDelCurso; // nombre del curso para este LibroCalificaciones
8
9     // método para establecer el nombre del curso
10    public void establecerNombreDelCurso( String nombre )
11    {
12       nombreDelCurso = nombre; // almacena el nombre del curso
13    } // fin del método establecerNombreDelCurso
14
15    // método para obtener el nombre del curso
16    public String obtenerNombreDelCurso()
17    {
18       return nombreDelCurso;
19    } // fin del método obtenerNombreDelCurso
20
21    // muestra un mensaje de bienvenida al usuario de LibroCalificaciones
22    public void mostrarMensaje()
23    {
24       // esta instrucción llama a obtenerNombreDelCurso para obtener el
25       // nombre del curso que representa este LibroCalificaciones
26       System.out.printf( "Bienvenido al libro de calificaciones para\n%s!\n",
27          obtenerNombreDelCurso() );
28    } // fin del método mostrarMensaje
29
30 } // fin de la clase LibroCalificaciones
```

Figura 3.7 | Clase `LibroCalificaciones` que contiene una variable de instancia `nombreDelCurso` y métodos para establecer y obtener su valor.

copia de cada variable de instancia. Por ejemplo, si hay dos objetos `LibroCalificaciones`, cada objeto tiene su propia copia de `nombreDelCurso` (una por cada objeto). Un beneficio de hacer de `nombreDelCurso` una variable de instancia es que todos los métodos de la clase (en este caso, `LibroCalificaciones`) pueden manipular cualquier variable de instancia que aparezca en la clase (en este caso, `nombreDelCurso`).

Los modificadores de acceso `public` *y* `private`

La mayoría de las declaraciones de variables de instancia van precedidas por la palabra clave `private` (como en la línea 7). Al igual que `public`, la palabra clave `private` es un modificador de acceso. Las variables o los métodos declarados con el modificador de acceso `private` son accesibles sólo para los métodos de la clase en la que se declaran. Así, la variable `nombreDelCurso` sólo puede utilizarse en los métodos `establecerNombreDelCurso`, `obtenerNombreDelCurso` y `mostrarMensaje` de (cada objeto de) la clase `LibroCalificaciones`.



Observación de ingeniería de software 3.3

Es necesario colocar un modificador de acceso antes de cada declaración de un campo y de un método. Como regla empírica, las variables de instancia deben declararse como `private` *y los métodos, como* `public`. *(Más adelante veremos que es apropiado declarar ciertos métodos como* `private`*, si sólo van a estar accesibles por otros métodos de la clase).*



Buena práctica de programación 3.1

Preferimos listar los campos de una clase primero, para que, a medida que usted lea el código, pueda ver los nombres y tipos de las variables antes de ver su uso en los métodos de la clase. Es posible listar los campos de la clase en cualquier parte de la misma, fuera de las declaraciones de sus métodos, pero si se esparcen por todo el código, éste será más difícil de leer.



Buena práctica de programación 3.2

Coloque una línea en blanco entre las declaraciones de los métodos, para separarlos y mejorar la legibilidad del programa.

El proceso de declarar variables de instancia con el modificador de acceso `private` se conoce como **ocultamiento de datos**. Cuando un programa crea (instancia) un objeto de la clase `LibroCalificaciones`, la variable `nombreDelCurso` se encapsula (oculta) en el objeto, y sólo está accesible para los métodos de la clase de ese objeto. En la clase `LibroCalificaciones`, los métodos `establecerNombreDelCurso` y `obtenerNombreDelCurso` manipulan a la variable de instancia `nombreDelCurso`.

El método `establecerNombreDelCurso` (líneas 10 a la 13) no devuelve datos cuando completa su tarea, por lo que su tipo de valor de retorno es `void`. El método recibe un parámetro (`nombre`), el cual representa el nombre del curso que se pasará al método como un argumento. La línea 12 asigna `nombre` a la variable de instancia `nombreDelCurso`.

El método `obtenerNombreDelCurso` (líneas 16 a la 19) devuelve un `nombreDelCurso` de un objeto `LibroCalificaciones` específico. El método tiene una lista de parámetros vacía, por lo que no requiere información adicional para realizar su tarea. El método especifica que devuelve un objeto `String`; a éste se le conoce como el **tipo de valor de retorno** del método. Cuando se hace una llamada a un método que especifica un tipo de valor de retorno, y éste completa su tarea, devuelve un resultado al método que lo llamó. Por ejemplo, cuando usted va a un cajero automático (ATM) y solicita el saldo de su cuenta, espera que el ATM le devuelva un valor que representa su saldo. De manera similar, cuando una instrucción llama al método `obtenerNombreDelCurso` en un objeto `LibroCalificaciones`, la instrucción espera recibir el nombre del curso de `LibroCalificaciones` (en este caso, un objeto `String`, como se especifica en el tipo de valor de retorno de la declaración del método). Si tiene un método `cuadrado` que devuelve el cuadrado de su argumento, es de esperarse que la instrucción

```
int resultado = cuadrado( 2 );
```

devuelva `4` del método `cuadrado` y asigne `4` a la variable `resultado`. Si tiene un método `maximo` que devuelve el mayor de tres argumentos enteros, es de esperarse que la siguiente instrucción

```
int mayor = maximo( 27, 114, 51 );
```

devuelva `114` del método `maximo` y asigne 114 a la variable `mayor`.

Observe que cada una de las instrucciones en las líneas 12 y 18 utilizan `nombreDelCurso`, aun cuando esta variable no se declaró en ninguno de los métodos. Podemos utilizar `nombreDelCurso` en los métodos de la clase `LibroCalificaciones`, ya que `nombreDelCurso` es un campo de la clase. Observe además que el orden en el que se declaran los métodos en una clase no determina cuándo se van a llamar en tiempo de ejecución. Por lo tanto, el método `obtenerNombreDelCurso` podría declararse antes del método `establecerNombreDelCurso`.

El método `mostrarMensaje` (líneas 22 a la 28) no devuelve datos cuando completa su tarea, por lo que su tipo de valor de retorno es `void`. El método no recibe parámetros, por lo que la lista de parámetros está vacía. Las líneas 26 y 27 imprimen un mensaje de bienvenida, que incluye el valor de la variable de instancia `nombreDelCurso`. Una vez más, necesitamos crear un objeto de la clase `LibroCalificaciones` y llamar a sus métodos para poder mostrar en pantalla el mensaje de bienvenida.

La clase `PruebaLibroCalificaciones` *que demuestra a la clase* `LibroCalificaciones`

La clase `PruebaLibroCalificaciones` (figura 3.8) crea un objeto de la clase `LibroCalificaciones` y demuestra el uso de sus métodos. La línea 11 crea un objeto `Scanner`, que se utilizará para obtener el nombre de un curso del usuario. La línea 14 crea un objeto `LibroCalificaciones` y lo asigna a la variable local `miLibroCalificaciones`, de tipo `LibroCalificaciones`. Las líneas 17-18 muestran el nombre inicial del curso mediante una llamada al método `obtenerNombreDelCurso` del objeto. Observe que la primera línea de la salida muestra el nombre "`null`". A diferencia de las variables locales, que no se inicializan de manera automática, cada campo tiene un **valor inicial predeterminado**: un valor que Java proporciona cuando el programador no especifica el valor inicial del campo. Por ende, no se requiere que los campos se inicialicen explícitamente antes de usarlos en un programa, a menos que deban inicializarse con valores distintos de los predeterminados. El valor predeterminado para un campo de tipo `String` (como `nombreDelCurso` en este ejemplo) es `null`, de lo cual hablaremos con más detalle en la sección 3.6.

La línea 21 pide al usuario que escriba el nombre para el curso. La variable `String` local `elNombre` (declarada en la línea 22) se inicializa con el nombre del curso que escribió el usuario, el cual se devuelve mediante la llamada al método `nextLine` del objeto `Scanner` llamado `entrada`. La línea 23 llama al método `establecerNombreDelCurso` del objeto `miLibroCalificaciones` y provee `elNombre` como argumento para el método. Cuando se hace la llamada al método, el valor del argumento se asigna al parámetro `nombre` (línea 10, figura 3.7) del método `establecerNombreDelCurso` (líneas 10 a la 13, figura 3.7). Después, el valor del parámetro se asigna a la variable de instancia `nombreDelCurso` (línea 12, figura 3.7). La línea 24 (figura 3.8) salta una línea en la salida, y después la línea 27 llama al método `mostrarMensaje` del objeto `miLibroCalificaciones` para mostrar en pantalla el mensaje de bienvenida, que contiene el nombre del curso.

Los métodos establecer *y* obtener

Los campos `private` de una clase pueden manipularse sólo mediante los métodos de esa clase. Por lo tanto, un **cliente de un objeto** (es decir, cualquier clase que llame a los métodos del objeto) llama a los métodos `public` de la clase para manipular los campos `private` de un objeto de esa clase. Esto explica por qué las instrucciones en el método `main` (figura 3.8) llaman a los métodos `establecerNombreDelCurso`, `obtenerNombreDelCurso` y `mostrarMensaje` en un objeto `LibroCalificaciones`. A menudo, las clases proporcionan métodos `public` para permitir a los clientes de la clase ***establecer*** (es decir, asignar valores a) u ***obtener*** (es decir, obtener los valores de) variables de instancia `private`. Los nombres de estos métodos no necesitan empezar con *establecer* u *obtener*, pero esta convención de nomenclatura es muy recomendada en Java, y es requerida para ciertos componentes de software especiales de Java, conocidos como JavaBeans, que pueden simplificar la programación en muchos entornos de desarrollo integrados (IDEs). El método que *establece* la variable `nombreDelCurso` en este ejemplo se llama `establecerNombreDelCurso`, y el método que *obtiene* el valor de la variable de instancia `nombreDelCurso` se llama `obtenerNombreDelCurso`.

```
1  // Fig. 3.8: PruebaLibroCalificaciones.java
2  // Crea y manipula un objeto LibroCalificaciones.
3  import java.util.Scanner; // el programa usa la clase Scanner
4
5  public class PruebaLibroCalificaciones
6  {
7     // el método main empieza la ejecución del programa
8     public static void main( String args[] )
9     {
10       // crea un objeto Scanner para obtener la entrada de la ventana de comandos
11       Scanner entrada = new Scanner( System.in );
12
13       // crea un objeto LibroCalificaciones y lo asigna a miLibroCalificaciones
14       LibroCalificaciones miLibroCalificaciones = new LibroCalificaciones();
15
16       // muestra el valor inicial de nombreDelCurso
17       System.out.printf( "El nombre inicial del curso es: %s\n\n",
18          miLibroCalificaciones.obtenerNombreDelCurso() );
19
20       // pide y lee el nombre del curso
21       System.out.println( "Escriba el nombre del curso:" );
22       String elNombre = entrada.nextLine(); // lee una línea de texto
23       miLibroCalificaciones.establecerNombreDelCurso( elNombre ); // establece el nombre
         del curso
24       System.out.println(); // imprime una línea en blanco
25
26       // muestra el mensaje de bienvenida después de especificar el nombre del curso
27       miLibroCalificaciones.mostrarMensaje();
28    } // fin de main
29
30 } // fin de la clase PruebaLibroCalificaciones
```

Figura 3.8 | Creación y manipulación de un objeto `LibroCalificaciones`. (Parte 1 de 2).

```
El nombre inicial del curso es: null

Escriba el nombre del curso:
CS101 Introduccion a la programacion en Java

Bienvenido al libro de calificaciones para
CS101 Introduccion a la programacion en Java!
```

Figura 3.8 | Creación y manipulación de un objeto `LibroCalificaciones`. (Parte 2 de 2).

Diagrama de clases de UML para la clase `LibroCalificaciones` *con una variable de instancia, y métodos* **establecer** *y* **obtener**

La figura 3.9 contiene un diagrama de clases de UML actualizado para la versión de la clase `LibroCalificaciones` de la figura 3.7. Este diagrama modela la variable de instancia `nombreDelCurso` de la clase `LibroCalificaciones` como un atributo en el compartimiento intermedio de la clase. UML representa a las variables de instancia como atributos, listando el nombre del atributo, seguido de dos puntos y del tipo del atributo. El tipo de UML del atributo `nombreDelCurso` es `String`. La variable de instancia `nombreDelCurso` es `private` en Java, por lo que el diagrama de clases lista un signo menos (-) en frente del nombre del atributo correspondiente. La clase `LibroCalificaciones` contiene tres métodos `public`, por lo que el diagrama de clases lista tres operaciones en el tercer compartimiento. Recuerde que el signo más (+) antes de cada nombre de operación indica que ésta es `public`. La operación `establecerNombreDelCurso` tiene un parámetro `String` llamado `nombre`. UML indica el tipo de valor de retorno de una operación colocando dos puntos y el tipo de valor de retorno después de los paréntesis que le siguen al nombre de la operación. El método `obtenerNombreDelCurso` de la clase `LibroCalificaciones` (figura 3.7) tiene un tipo de valor de retorno `String` en Java, por lo que el diagrama de clases muestra un tipo de valor de retorno `String` en UML. Observe que las operaciones `establecerNombreDelCurso` y `mostrarMensaje` no devuelven valores (es decir, devuelven `void` en Java), por lo que el diagrama de clases de UML no especifica un tipo de valor de retorno después de los paréntesis de estas operaciones.

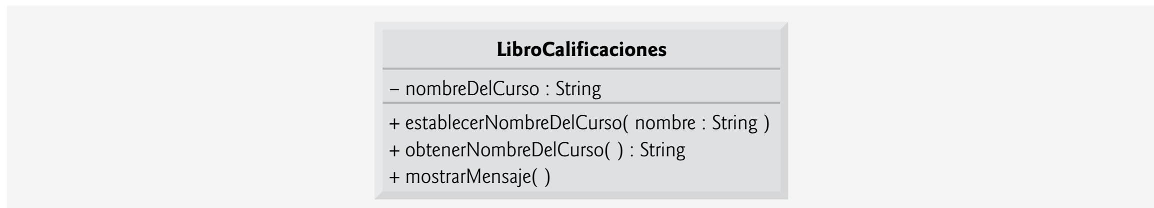


Figura 3.9 | Diagrama de clases de UML, en el que se indica que la clase `LibroCalificaciones` tiene un atributo `nombreDelCurso` de tipo `String` en UML, y tres operaciones: `establecerNombreDelCurso` (con un parámetro `nombre` de tipo `String` de UML), `obtenerNombreDelCurso` (que devuelve el tipo `String` de UML) y `mostrarMensaje`.

3.6 Comparación entre tipos primitivos y tipos por referencia

Los tipos de datos en Java se dividen en dos categorías: tipos primitivos y **tipos por referencia** (algunas veces conocidos como **tipos no primitivos**). Los tipos primitivos son `boolean, byte, char, short, int, long, float` y `double`. Todos los tipos no primitivos son tipos por referencia, por lo cual las clases, que especifican los tipos de objetos, son tipos por referencia.

Una variable de tipo primitivo puede almacenar sólo un valor de su tipo declarado a la vez. Por ejemplo, una variable `int` puede almacenar un número completo (como 7) a la vez. Cuando se asigna otro valor a esa variable, se sustituye su valor inicial. Las variables de instancia de tipo primitivo se inicializan de manera predeterminada; las variables de los tipos `byte, char, short, int, long, float` y `double` se inicializan con 0, y las variables de tipo `boolean` se inicializan con `false`. Usted puede especificar sus propios valores iniciales para las variables de tipo primitivo. Recuerde que las variables locales *no* se inicializan de manera predeterminada.



Tip para prevenir errores 3.1

Cualquier intento de utilizar una variable local que no se haya inicializado produce un error de compilación.

Los programas utilizan variables de tipo por referencia (que por lo general se llaman **referencias**) para almacenar las ubicaciones de los objetos en la memoria de la computadora. Se dice que dicha variable hace **referencia a un objeto** en el programa. Cada uno de los objetos a los que se hace referencia pueden contener muchas variables de instancia y métodos. La línea 14 de la figura 3.8 crea un objeto de la clase `LibroCalificaciones`, y la variable `miLibroCalificaciones` contiene una referencia a ese objeto `LibroCalificaciones`. Las variables de instancia de tipo por referencia se inicializan de manera predeterminada con el valor `null`: una palabra reservada que representa una "referencia a nada". Esto explica por qué la primera llamada a `obtenerNombreDelCurso` en la línea 18 de la figura 3.8 devolvía `null`; no se había establecido el valor de `nombreDelCurso`, por lo que se devolvía el valor inicial predeterminado `null`. En el apéndice C, Palabras clave y palabras reservadas, se muestra una lista completa de las palabras reservadas y las palabras clave.

Es obligatorio que una referencia a un objeto **invoque** (es decir, llame) a los métodos de un objeto. En la aplicación de la figura 3.8, las instrucciones en el método `main` utilizan la variable `miLibroCalificaciones` para enviar mensajes al objeto `LibroCalificaciones`. Estos mensajes son llamadas a métodos (como `establecerNombreDelCurso` y `obtenerNombreDelCurso`) que permiten al programa interactuar con el objeto `LibroCalificaciones`. Por ejemplo, la instrucción en la línea 23 utiliza a `miLibroCalificaciones` para enviar el mensaje `establecerNombreDelCurso` al objeto `LibroCalificaciones`. El mensaje incluye el argumento que requiere `establecerNombreDelCurso` para realizar su tarea. El objeto `LibroCalificaciones` utiliza esta información para establecer la variable de instancia `nombreDelCurso`. Tenga en cuenta que las variables de tipo primitivo no hacen referencias a objetos, por lo que dichas variables no pueden utilizarse para invocar métodos.



Observación de ingeniería de software 3.4

El tipo declarado de una variable (por ejemplo, `int`, `double` *o* `LibroCalificaciones`*) indica si la variable es de tipo primitivo o por referencia. Si el tipo de una variable no es uno de los ocho tipos primitivos, entonces es un tipo por referencia. (Por ejemplo,* `Cuenta cuenta1` *indica que* `cuenta1` *es una referencia a un objeto* `Cuenta`*).*

3.7 Inicialización de objetos mediante constructores

Como mencionamos en la sección 3.5, cuando se crea un objeto de la clase `LibroCalificaciones` (figura 3.7), su variable de instancia `nombreCurso` se inicializa con `null` de manera predeterminada. ¿Qué pasa si usted desea proporcionar el nombre de un curso a la hora de crear un objeto `LibroCalificaciones`? Cada clase que usted declare puede proporcionar un constructor, el cual puede utilizarse para inicializar un objeto de una clase al momento de crear ese objeto. De hecho, Java requiere una llamada al constructor para cada objeto que se crea. La palabra clave `new` llama al constructor de la clase para realizar la inicialización. La llamada al constructor se indica mediante el nombre de la clase, seguido de paréntesis; el constructor *debe* tener el mismo nombre que la clase. Por ejemplo, la línea 14 de la figura 3.8 primero utiliza `new` para crear un objeto `LibroCalificaciones`. Los paréntesis vacíos después de `"new LibroCalificaciones"` indican una llamada sin argumentos al constructor de la clase. De manera predeterminada, el compilador proporciona un **constructor predeterminado** sin parámetros, en cualquier clase que no incluya un constructor en forma explícita. Cuando una clase sólo tiene el constructor predeterminado, sus variables de instancia se inicializan con sus valores predeterminados. Las variables de los tipos `char`, `byte`, `short`, `int`, `long`, `float` y `double` se inicializan con `0`, las variables de tipo `boolean` se inicializan con `false`, y las variables de tipo por referencia se inicializan con `null`.

Cuando usted declara una clase, puede proporcionar su propio constructor para especificar una inicialización personalizada para los objetos de su clase. Por ejemplo, tal vez un programador quiera especificar el nombre de un curso para un objeto `LibroCalificaciones` cuando se crea este objeto, como en

```
LibroCalificaciones miLibroCalificaciones =
   new LibroCalificaciones( "CS101 Introduccion a la programacion en Java" );
```

En este caso, el argumento `"CS101 Introduccion a la programacion en Java"` se pasa al constructor del objeto `LibroCalificaciones` y se utiliza para inicializar el `nombreDelCurso`. La instrucción anterior requiere que la clase proporcione un constructor con un parámetro `String`. La figura 3.10 contiene una clase `LibroCalificaciones` modificada con dicho constructor.

```
1  // Fig. 3.10: LibroCalificaciones.java
2  // La clase LibroCalificaciones con un constructor para inicializar el nombre del curso.
3
4  public class LibroCalificaciones
5  {
6     private String nombreDelCurso; // nombre del curso para este LibroCalificaciones
7
8     // el constructor inicializa nombreDelCurso con el objeto String que se provee como
      argumento
9     public LibroCalificaciones( String nombre )
10    {
11       nombreDelCurso = nombre; // inicializa nombreDelCurso
12    } // fin del constructor
13
14    // método para establecer el nombre del curso
15    public void establecerNombreDelCurso( String nombre )
16    {
17       nombreDelCurso = nombre; // almacena el nombre del curso
18    } // fin del método establecerNombreDelCurso
19
20    // método para obtener el nombre del curso
21    public String obtenerNombreDelCurso()
22    {
23       return nombreDelCurso;
24    } // fin del método obtenerNombreDelCurso
25
26    // muestra un mensaje de bienvenida al usuario de LibroCalificaciones
27    public void mostrarMensaje()
28    {
29       // esta instrucción llama a obtenerNombreDelCurso para obtener el
30       // nombre del curso que este LibroCalificaciones representa
31       System.out.printf( "Bienvenido al Libro de calificaciones para\n%s!\n",
32          obtenerNombreDelCurso() );
33    } // fin del método mostrarMensaje
34
35 } // fin de la clase LibroCalificaciones
```

Figura 3.10 | La clase `LibroCalificaciones` con un constructor para inicializar el nombre del curso.

Las líneas 9 a la 12 declaran el constructor para la clase `LibroCalificaciones`. Un constructor debe tener el mismo nombre que su clase. Al igual que un método, un constructor especifica en su lista de parámetros los datos que requiere para realizar su tarea. Cuando usted crea un nuevo objeto (como haremos en la figura 3.11), estos datos se colocan en los paréntesis que van después del nombre de la clase. La línea 9 indica que el constructor de la clase `LibroCalificaciones` tiene un parámetro `String` llamado `nombre`. El `nombre` que se pasa al constructor se asigna a la variable de instancia `nombreCurso` en la línea 11 del cuerpo del constructor.

La figura 3.11 demuestra la inicialización de objetos `LibroCalificaciones` mediante el uso de este constructor. Las líneas 11 y 12 crean e inicializan el objeto `libroCalificaciones1` de `LibroCalificaciones`. El constructor de la clase `LibroCalificaciones` se llama con el argumento `"CS101 Introduccion a la programacion en Java"` para inicializar el nombre del curso. La expresión de creación de la instancia de la clase a la derecha del signo = en las líneas 11 y 12 devuelve una referencia al nuevo objeto, el cual se asigna a la variable `libroCalificaciones1`. Las líneas 13 y 14 repiten este proceso para otro objeto `LibroCalificaciones` llamado `libroCalificaciones2`, pero esta vez se le pasa el argumento `"CS102 Estructuras de datos en Java"` para inicializar el nombre del curso para `libroCalificaciones2`. Las líneas 17 a la 20 utilizan el método *`obtenerNombreDelCurso`* de cada objeto para obtener los nombres de los cursos y mostrar que, sin duda, se inicializaron en el momento en el que se crearon los objetos. En la introducción a la sección 3.5, aprendió que cada instancia (es decir, objeto) de una clase contiene su propia copia de las variables de instancia de la clase. La salida confirma que cada objeto `LibroCalificaciones` mantiene su propia copia de la variable de instancia `nombreCurso`.

```java
// Fig. 3.11: PruebaLibroCalificaciones.java
// El constructor de LibroCalificaciones se utiliza para especificar el
// nombre del curso cada vez que se crea cada objeto LibroCalificaciones.

public class PruebaLibroCalificaciones
{
   // el método main empieza la ejecución del programa
   public static void main( String args[] )
   {
      // crea objeto LibroCalificaciones
      LibroCalificaciones libroCalificaciones1 = new LibroCalificaciones(
         "CS101 Introduccion a la programacion en Java" );
      LibroCalificaciones libroCalificaciones2 = new LibroCalificaciones(
         "CS102 Estructuras de datos en Java" );

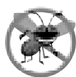
      // muestra el valor inicial de nombreDelCurso para cada LibroCalificaciones
      System.out.printf( "El nombre del curso de libroCalificaciones1 es: %s\n",
         libroCalificaciones1.obtenerNombreDelCurso() );
      System.out.printf( "El nombre del curso de libroCalificaciones2 es: %s\n",
         libroCalificaciones2.obtenerNombreDelCurso() );
   } // fin de main

} // fin de la clase PruebaLibroCalificaciones
```

```
El nombre del curso de libroCalificaciones1 es: CS101 Introduccion a la programacion en Java
El nombre del curso de libroCalificaciones2 es: CS102 Estructuras de datos en Java
```

Figura 3.11 | El constructor de `LibroCalificaciones` se utiliza para especificar el nombre del curso cada vez que se crea un objeto `LibroCalificaciones`.

Al igual que los métodos, los constructores también pueden recibir argumentos. No obstante, una importante diferencia entre los constructores y los métodos es que los constructores no pueden devolver valores, por lo cual no pueden especificar un tipo de valor de retorno (ni siquiera `void`). Por lo general, los constructores se declaran como `public`. Si una clase no incluye un constructor, las variables de instancia de esa clase se inicializan con sus valores predeterminados. Si un programador declara uno o más constructores para una clase, el compilador de Java no creará un constructor predeterminado para esa clase.



Tip para prevenir errores 3.2

A menos que sea aceptable la inicialización predeterminada de las variables de instancia de su clase, deberá proporcionar un constructor para asegurarse que las variables de instancia de su clase se inicialicen en forma apropiada con valores significativos, a la hora de crear cada nuevo objeto de su clase.

Agregar el constructor al diagrama de clases de UML de la clase `LibroCalificaciones`

El diagrama de clases de UML de la figura 3.12 modela la clase `LibroCalificaciones` de la figura 3.10, la cual tiene un constructor con un parámetro llamado `nombre`, de tipo `String`. Al igual que las operaciones, en un diagrama de clases, UML modela a los constructores en el tercer compartimiento de una clase. Para diferenciar a un constructor de las operaciones de una clase, UML requiere que se coloque la palabra "constructor" entre los signos « y » antes del nombre del constructor. Es costumbre listar los constructores antes de otras operaciones en el tercer compartimiento.

3.8 Números de punto flotante y el tipo double

En nuestra siguiente aplicación, dejaremos por un momento nuestro ejemplo práctico con la clase `LibroCalificaciones` para declarar una clase llamada `Cuenta`, la cual mantiene el saldo de una cuenta bancaria. La mayoría de los saldos de las cuentas no son números enteros (por ejemplo, 0, –22 y 1024). Por esta razón, la clase `Cuenta`

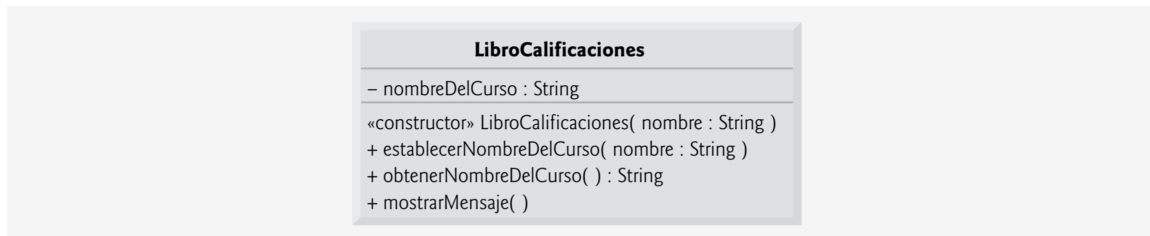


Figura 3.12 | Diagrama de clases de UML, en el cual se indica que la clase `LibroCalificaciones` tiene un constructor con un parámetro `nombre` del tipo `String` de UML.

representa el saldo de las cuentas como un **número de punto flotante** (es decir, un número con un punto decimal, como 7.33, 0.0975 o 1000.12345). Java cuenta con dos tipos primitivos para almacenar números de punto flotante en la memoria: `float` y `double`. La principal diferencia entre ellos es que las variables tipo `double` pueden almacenar números con mayor magnitud y detalle (es decir, más dígitos a la derecha del punto decimal; lo que también se conoce como **precisión** del número) que las variables `float`.

Precisión de los números de punto flotante y requerimientos de memoria

Las variables de tipo `float` representan **números de punto flotante de precisión simple** y tienen siete dígitos significativos. Las variables de tipo `double` representan **números de punto flotante de precisión doble**. Éstos requieren el doble de memoria que las variables `float` y proporcionan 15 dígitos significativos; aproximadamente el doble de precisión de las variables `float`. Para el rango de valores requeridos por la mayoría de los programas, debe bastar con las variables de tipo `float`, pero podemos utilizar variables tipo `double` para "ir a la segura". En algunas aplicaciones, incluso hasta las variables de tipo `double` serán inadecuadas; dichas aplicaciones se encuentran más allá del alcance de este libro. La mayoría de los programadores representan los números de punto flotante con el tipo `double`. De hecho, Java trata a todos los números de punto flotante que escribimos en el código fuente de un programa (como 7.33 y 0.0975) como valores `double` de manera predeterminada. Dichos valores en el código fuente se conocen como **literales de punto flotante**. En el apéndice D, Tipos primitivos, podrá consultar los rangos de los valores para los tipos `float` y `double`.

Aunque los números de punto flotante no son siempre 100% precisos, tienen numerosas aplicaciones. Por ejemplo, cuando hablamos de una temperatura corporal "normal" de 36.8, no necesitamos una precisión con un número extenso de dígitos. Cuando leemos la temperatura en un termómetro como 36.8, en realidad podría ser 36.7999473210643. Si consideramos a este número simplemente como 36.8, está bien para la mayoría de las aplicaciones en las que se trabaja con las temperaturas corporales. Debido a la naturaleza imprecisa de los números de punto flotante, se prefiere el tipo `double` al tipo `float` ya que las variables `double` pueden representar números de punto flotante con más precisión. Por esta razón, utilizaremos el tipo `double` a lo largo de este libro.

Los números de punto flotante también surgen como resultado de la división. En la aritmética convencional, cuando dividimos 10 entre 3 el resultado es 3.3333333…, y la secuencia de números 3 se repite en forma indefinida. La computadora asigna sólo una cantidad fija de espacio para almacenar un valor de este tipo, por lo que, sin duda, el valor de punto flotante almacenado sólo puede ser una aproximación.



Error común de programación 3.4

El uso de números de punto flotante en una forma en la que se asuma que se representan con precisión puede producir errores lógicos.

La clase `Cuenta` con una variable de instancia de tipo `double`

Nuestra siguiente aplicación (figuras 3.13 y 3.14) contiene una clase llamada `Cuenta` (figura 3.13), la cual mantiene el saldo de una cuenta bancaria. Un banco ordinario da servicio a muchas cuentas, cada una con su propio saldo, por lo que la línea 7 declara una variable de instancia, de tipo `double`, llamada `saldo`. La variable `saldo` es una variable de instancia, ya que está declarada en el cuerpo de la clase pero fuera de las declaraciones de los métodos de la misma (líneas 10 a la 16, 19 a la 22 y 25 a la 28). Cada instancia (es decir, objeto) de la clase `Cuenta` contiene su propia copia de `saldo`.

```
1 // Fig. 3.13: Cuenta.java
2 // La clase Cuenta con un constructor para
3 // inicializar la variable de instancia saldo.
4
5 public class Cuenta
6 {
7    private double saldo; // variable de instancia que almacena el saldo
8
9    // constructor
10   public Cuenta( double saldoInicial )
11   {
12      // valida que saldoInicial sea mayor que 0.0;
13      // si no lo es, saldo se inicializa con el valor predeterminado 0.0
14      if ( saldoInicial > 0.0 )
15         saldo = saldoInicial;
16   } // fin del constructor de Cuenta
17
18   // abona (suma) un monto a la cuenta
19   public void abonar( double monto )
20   {
21      saldo = saldo + monto; // suma el monto al saldo
22   } // fin del método abonar
23
24   // devuelve el saldo de la cuenta
25   public double obtenerSaldo()
26   {
27      return saldo; // proporciona el valor de saldo al método que hizo la llamada
28   } // fin del método obtenerSaldo
29
30 } // fin de la clase Cuenta
```

Figura 3.13 | La clase `Cuenta` con una variable de instancia de tipo `double`.

La clase `Cuenta` contiene un constructor y dos métodos. Debido a que es común que alguien abra una cuenta para depositar dinero de inmediato, el constructor (líneas 10 a la 16) recibe un parámetro llamado `saldoInicial` de tipo `double`, el cual representa el saldo inicial de la cuenta. Las líneas 14 y 15 aseguran que `saldoInicial` sea mayor que `0.0`. De ser así, el valor de `saldoInicial` se asigna a la variable de instancia `saldo`. En caso contrario, `saldo` permanece en `0.0`, su valor inicial predeterminado.

El método `abonar` (líneas 19 a la 22) no devuelve datos cuando completa su tarea, por lo que su tipo de valor de retorno es `void`. El método recibe un parámetro llamado `monto`: un valor `double` que se sumará al saldo. La línea 21 suma `monto` al valor actual de `saldo`, y después asigna el resultado a `saldo` (con lo cual se sustituye el monto del saldo anterior).

El método `obtenerSaldo` (líneas 25 a la 28) permite a los clientes de la clase (es decir, otras clases que utilicen esta clase) obtener el valor del `saldo` de un objeto `Cuenta` específico. El método especifica el tipo de valor de retorno `double` y una lista de parámetros vacía.

Observe una vez más que las instrucciones en las líneas 15, 21 y 27 utilizan la variable de instancia `saldo`, aun y cuando no se declaró en ninguno de los métodos. Podemos usar `saldo` en estos métodos, ya que es una variable de instancia de la clase.

La clase `PruebaCuenta` *que utiliza a la clase* `Cuenta`

La clase `PruebaCuenta` (figura 3.14) crea dos objetos `Cuenta` (líneas 10 y 11) y los inicializa con `50.00` y `-7.53`, respectivamente. Las líneas 14 a la 17 imprimen el saldo en cada objeto `Cuenta` mediante una llamada al método `obtenerSaldo` de `Cuenta`. Cuando se hace una llamada al método `obtenerSaldo` para `cuenta1` en la línea 15, se devuelve el valor del saldo de `cuenta1` de la línea 27 en la figura 3.13, y se imprime en pantalla mediante la instrucción `System.out.printf` (figura 3.14, líneas 14 y 15). De manera similar, cuando se hace la llamada al método `obtenerSaldo` para `cuenta2` en la línea 17, se devuelve el valor del saldo de `cuenta2` de la línea 27 en la

figura 3.13, y se imprime en pantalla mediante la instrucción `System.out.printf` (figura 3.14, líneas 16 y 17). Observe que el saldo de `cuenta2` es `0.00`, ya que el constructor se aseguró de que la cuenta no pudiera empezar con un saldo negativo. El valor se imprime en pantalla mediante `printf`, con el especificador de formato `%.2f`. El especificador de formato **%f** se utiliza para imprimir valores de tipo `float` o `double`. El `.2` entre `%` y `f` representa el número de lugares decimales (2) que deben imprimirse a la derecha del punto decimal en el número de punto flotante; a esto también se le conoce como la **precisión** del número. Cualquier valor de punto flotante que se imprima con `%.2f` se redondeará a la posición de las centenas; por ejemplo, 123.457 se redondearía a 123.46, y 27.333 se redondearía a 27.33.

```
1  // Fig. 3.14: PruebaCuenta.java
2  // Entrada y salida de números de punto flotante con objetos Cuenta.
3  import java.util.Scanner;
4
5  public class PruebaCuenta
6  {
7     // el método main empieza la ejecución de la aplicación de Java
8     public static void main( String args[] )
9     {
10       Cuenta cuenta1 = new Cuenta( 50.00 ); // crea objeto Cuenta
11       Cuenta cuenta2 = new Cuenta( -7.53 ); // crea objeto Cuenta
12
13       // muestra el saldo inicial de cada objeto
14       System.out.printf( “Saldo de cuenta1: $%.2f\n”,
15          cuenta1.obtenerSaldo() );
16       System.out.printf( “Saldo de cuenta2: $%.2f\n\n”,
17          cuenta2.obtenerSaldo() );
18
19       // crea objeto Scanner para obtener la entrada de la ventana de comandos
20       Scanner entrada = new Scanner( System.in );
21       double montoDeposito; // deposita el monto escrito por el usuario
22
23       System.out.print( “Escriba el monto a depositar para cuenta1: “ ); // indicador
24       montoDeposito = entrada.nextDouble(); // obtiene entrada del usuario
25       System.out.printf( “\nsumando %.2f al saldo de cuenta1\n\n”,
26          montoDeposito );
27       cuenta1.abonar( montoDeposito ); // suma al saldo de cuenta1
28
29       // muestra los saldos
30       System.out.printf( “Saldo de cuenta1: $%.2f\n”,
31          cuenta1.obtenerSaldo() );
32       System.out.printf( “Saldo de cuenta2: $%.2f\n\n”,
33          cuenta2.obtenerSaldo() );
34
35       System.out.print( “Escriba el monto a depositar para cuenta2: “ ); // indicador
36       montoDeposito = entrada.nextDouble(); // obtiene entrada del usuario
37       System.out.printf( “\nsumando %.2f al saldo de cuenta2\n\n”,
38          montoDeposito );
39       cuenta2.abonar( montoDeposito ); // suma al saldo de cuenta2
40
41       // muestra los saldos
42       System.out.printf( “Saldo de cuenta1: $%.2f\n”,
43          cuenta1.obtenerSaldo() );
44       System.out.printf( “Saldo de cuenta2: $%.2f\n”,
45          cuenta2.obtenerSaldo() );
46    } // fin de main
47
48 } // fin de la clase PruebaCuenta
```

Figura 3.14 | Entrada y salida de números de punto flotante con objetos `Cuenta`. (Parte 1 de 2).

```
Saldo de cuenta1: $50.00
Saldo de cuenta2: $0.00

Escriba el monto a depositar para cuenta1: 25.53

sumando 25.53 al saldo de cuenta1

Saldo de cuenta1: $75.53
Saldo de cuenta2: $0.00

Escriba el monto a depositar para cuenta2: 123.45

sumando 123.45 al saldo de cuenta2

Saldo de cuenta1: $75.53
Saldo de cuenta2: $123.45
```

Figura 3.14 | Entrada y salida de números de punto flotante con objetos `Cuenta`. (Parte 2 de 2).

La línea 20 crea un objeto `Scanner`, el cual se utilizará para obtener montos de depósito de un usuario. La línea 21 declara la variable local `montoDeposito` para almacenar cada monto de depósito introducido por el usuario. A diferencia de la variable de instancia `saldo` en la clase `Cuenta`, la variable local `montoDeposito` en `main` no se inicializa con `0.0` de manera predeterminada. Sin embargo, esta variable no necesita inicializarse aquí, ya que su valor se determinará con base a la entrada del usuario.

La línea 23 pide al usuario que escriba un monto a depositar para `cuenta1`. La línea 24 obtiene la entrada del usuario, llamando al método `nextDouble` del objeto `Scanner` llamado `entrada`, el cual devuelve un valor `double` introducido por el usuario. Las líneas 25 y 26 muestran el monto del depósito. La línea 27 llama al método `abonar` del objeto `cuenta1` y le suministra `montoDeposito` como argumento. Cuando se hace la llamada al método, el valor del argumento se asigna al parámetro `monto` (línea 19 de la figura 3.13) del método `abonar` (líneas 19 a la 22 de la figura 3.13), y después el método `abonar` suma ese valor al `saldo` (línea 21 de la figura 3.13). Las líneas 30 a la 33 (figura 3.14) imprimen en pantalla los saldos de ambos objetos `Cuenta` otra vez, para mostrar que sólo se modificó el saldo de `cuenta1`.

La línea 35 pide al usuario que escriba un monto a depositar para `cuenta2`. La línea 36 obtiene la entrada del usuario, llamando al método `nextDouble` del objeto `Scanner` llamado `entrada`. Las líneas 37 y 38 muestran el monto del depósito. La línea 39 llama al método `abonar` del objeto `cuenta2` y le suministra `montoDeposito` como argumento; después, el método `abonar` suma ese valor al saldo. Por último, las líneas 42 a la 45 imprimen en pantalla los saldos de ambos objetos `Cuenta` otra vez, para mostrar que sólo se modificó el saldo de `cuenta2`.

Diagrama de clases de UML para la clase `Cuenta`

El diagrama de clases de UML en la figura 3.15 modela la clase `Cuenta` de la figura 3.13. El diagrama modela la propiedad `private` llamada `saldo` con el tipo `Double` de UML, para que corresponda a la variable de instancia `saldo` de la clase, que tiene el tipo `double` de Java. El diagrama modela el constructor de la clase `Cuenta` con un parámetro `saldoInicial` del tipo `Double` de UML en el tercer compartimiento de la clase. Los dos métodos `public` de la clase se modelan como operaciones en el tercer compartimiento también. El diagrama modela la operación `abonar` con un parámetro `monto` de tipo `Double` de UML (ya que el método correspondiente tiene un parámetro `monto` de tipo `double` en Java) y la operación `obtenerSaldo` con un tipo de valor de retorno `Double` (ya que el método correspondiente en Java devuelve un valor `double`).

3.9 (Opcional) Ejemplo práctico de GUI y gráficos: uso de cuadros de diálogo

Este ejemplo práctico opcional está diseñado para aquellos quienes desean empezar a conocer las poderosas herramientas de Java para crear interfaces gráficas de usuario (GUIs) y gráficos antes de las principales discusiones de estos temas en el capítulo 11, Componentes de la GUI: parte 1, el capítulo 12, Gráficos y Java 2D™, y el capítulo 22, Componentes de la GUI: parte 2.

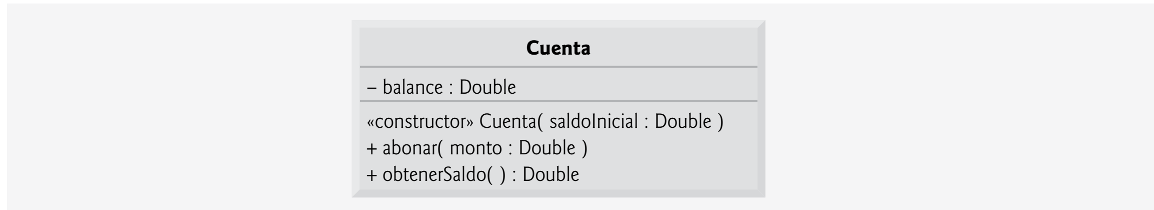


Figura 3.15 | Diagrama de clases de UML, el cual indica que la clase `Cuenta` tiene un atributo `private llamado saldo`, con el tipo `Double` de UML, un constructor (con un parámetro de tipo `Double` de UML) y dos `operaciones public`: `abonar` (con un parámetro `monto` de tipo `Double` de UML) y `obtenerSaldo` (devuelve el tipo `Double` de UML).

El ejemplo práctico de GUI y gráficos aparece en 10 secciones breves (figura 3.16). Cada sección introduce unos cuantos conceptos básicos y proporciona ejemplos visuales y gráficos. En las primeras secciones, creará sus primeras aplicaciones gráficas; en las secciones posteriores, utilizará los conceptos de programación orientada a objetos que se presentan a lo largo del capítulo 10 para crear una aplicación de dibujo, la cual dibuja una variedad de figuras. Cuando presentemos formalmente a las GUIs en el capítulo 11, utilizaremos el ratón para elegir exactamente qué figuras dibujar y en dónde dibujarlas. En el capítulo 12, agregaremos las herramientas de la API de gráficos en 2D de Java para dibujar las figuras con distintos grosores de línea y rellenos. Esperamos que este ejemplo práctico le sea informativo y divertido.

Ubicación	Título – Ejercicio(s)
Sección 3.9	Uso de cuadros de diálogo: entrada y salida básica con cuadros de diálogo.
Sección 4.14	Creación de dibujos simples: mostrar y dibujar líneas en la pantalla.
Sección 5.10	Dibujo de rectángulos y óvalos: uso de figuras para representar datos.
Sección 6.13	Colores y figuras rellenas: dibujar un tiro al blanco y gráficos aleatorios.
Sección 7.13	Dibujo de arcos: dibujar espirales con arcos.
Sección 8.18	Uso de objetos con gráficos: almacenar figuras como objetos.
Sección 9.8	Mostrar texto e imágenes mediante el uso de etiquetas: proporcionar información de estado.
Sección 10.8	Dibujo con polimorfismo: identificar las similitudes entre figuras.
Ejercicio 11.18	Expansión de la interfaz: uso de componentes de la GUI y manejo de eventos.
Ejercicio 12.31	Agregar Java 2D: uso de la API 2D de Java para mejorar los dibujos.

Figura 3.16 | Glosario de GUI ejemplo práctico en cada capítulo.

Cómo mostrar texto en un cuadro de diálogo

Los programas que hemos presentado hasta ahora muestran su salida en la ventana de comandos. Muchas aplicaciones utilizan ventanas, o **cuadros de diálogo** (también llamados **diálogos**) para mostrar la salida. Por ejemplo, los navegadores Web como Firefox o Microsoft Internet Explorer muestran las páginas Web en sus propias ventanas. Los programas de correo electrónico le permiten escribir y leer mensajes en una ventana. Por lo general, los cuadros de diálogo son ventanas en las que los programas muestran mensajes importantes a los usuarios. La clase **`JOptionPane`** cuenta con cuadros de diálogo previamente empaquetados, los cuales permiten a los programas mostrar ventanas que contengan mensajes; a dichas ventanas se les conoce como **diálogos de mensaje**. La figura 3.17 muestra el objeto `String "Bienvenido\na\nJava"` en un diálogo de mensaje.

La línea 3 indica que el programa utiliza la clase `JOptionPane` del paquete `javax.swing`. Este paquete contiene muchas clases que le ayudan a crear **interfaces gráficas de usuario (GUIs)** para las aplicaciones. Los **componentes de la GUI** facilitan la entrada de datos al usuario del programa, y la presentación de los datos de salida. La línea 10 llama al método **`showMessageDialog`** de `JOptionPane` para mostrar un cuadro de diálogo que contiene un mensaje. El método requiere dos argumentos. El primero ayuda a Java a determinar en dónde

```
// Fig. 3.17: Dialogo1.java
// Imprimir varias líneas en un cuadro de diálogo.
import javax.swing.JOptionPane; // importa la clase JOptionPane

public class Dialogo1
{
   public static void main( String args[] )
   {
      // muestra un cuadro de diálogo con un mensaje
      JOptionPane.showMessageDialog( null, "Bienvenido\na\nJava" );
   } // fin de main
} // fin de la clase Dialogo1
```



Figura 3.17 | Uso de `JOptionPane` para mostrar varias líneas en un cuadro de diálogo.

colocar el cuadro de diálogo. Cuando el primer argumento es `null`, el cuadro de diálogo aparece en el centro de la pantalla de la computadora. El segundo argumento es el objeto `String` a mostrar en el cuadro de diálogo.

El método `showMessageDialog` es un **método `static`** de la clase `JOptionPane`. A menudo, los métodos `static` definen las tareas utilizadas con frecuencia, y no se requiere crear explícitamente un objeto. Por ejemplo, muchos programas muestran cuadros de diálogo. En vez de que usted tenga que crear código para realizar esta tarea, los diseñadores de la clase `JOptionPane` de Java declaran un método `static` que realiza esta tarea por usted. Por lo general, la llamada a un método `static` se realiza mediante el uso del nombre de su clase, seguido de un punto (`.`) y del nombre del método, como en

NombreClase.*nombreMétodo*(*argumentos*)

El capítulo 6, Métodos: un análisis más detallado, habla sobre los métodos `static` con detalle.

Introducir texto en un cuadro de diálogo

La aplicación de la figura 3.18 utiliza otro cuadro de diálogo `JOptionPane` predefinido, conocido como **diálogo de entrada**, el cual permite al usuario introducir datos en el programa. El programa pide el nombre del usuario, y responde con un diálogo de mensaje que contiene un saludo y el nombre introducido por el usuario.

```
// Fig. 3.18: DialogoNombre.java
// Entrada básica con un cuadro de diálogo.
import javax.swing.JOptionPane;

public class DialogoNombre
{
   public static void main( String args[] )
   {
      // pide al usuario que escriba su nombre
      String nombre =
         JOptionPane.showInputDialog( "Cual es su nombre?" );

      // crea el mensaje
      String mensaje =
```

Figura 3.18 | Cómo obtener la entrada del usuario mediante un cuadro de diálogo. (Parte 1 de 2).

```
15          String.format( "Bienvenido, %s, a la programacion en Java!", nombre );
16
17       // muestra el mensaje para dar la bienvenida al usuario por su nombre
18       JOptionPane.showMessageDialog( null, mensaje );
19    } // fin de main
20 } // fin de la clase DialogoNombre
```



Figura 3.18 | Cómo obtener la entrada del usuario mediante un cuadro de diálogo. (Parte 2 de 2).

Las líneas 10 y 11 utilizan el método **`showInputDialog`** de `JOptionPane` para mostrar un diálogo de entrada que contiene un indicador y un campo (conocido como **campo de texto**), en el cual el usuario puede escribir texto. El argumento de `showInputDialog` es el indicador que muestra lo que el usuario debe escribir. El usuario escribe caracteres en el campo de texto, y después hace clic en el botón **Aceptar** u oprime la tecla *Intro* para devolver el objeto `String` al programa. El método `showInputDialog` (línea 11) devuelve un objeto `String` que contiene los caracteres escritos por el usuario. Almacenamos el objeto `String` en la variable `nombre` (línea 10). [*Nota:* si oprime el botón **Cancelar** en el cuadro de diálogo, el método devuelve `null` y el programa muestra la palabra clave "null" como el nombre].

Las líneas 14 y 15 utilizan el método `static String` llamado **`format`** para devolver un objeto `String` que contiene un saludo con el nombre del usuario. El método format es similar al método `System.out.printf`, excepto que `format` devuelve el objeto `String` con formato, en vez de mostrarlo en una ventana de comandos. La línea 18 muestra el saludo en un cuadro de diálogo de mensaje.

Ejercicio del ejemplo práctico de GUI y gráficos

3.1 Modifique el programa de suma en la figura 2.7 para usar la entrada y salida basadas en cuadro de diálogo con los métodos de la clase `JOptionPane`. Como el método `showInputDialog` devuelve un objeto `String`, debe convertir el objeto `String` que introduce el usuario a un `int` para usarlo en los cálculos. El método

```
Integer.parseInt( String s )
```

toma un argumento `String` que representa a un entero (por ejemplo, el resultado de `JOptionPane.showInputDialog`) y devuelve el valor como un `int`. El método `parseInt` es un método `static` de la clase `Integer` (del paquete `java.lang`). Observe que si el objeto `String` no contiene un entero válido, el programa terminará con un error.

3.10 (Opcional) Ejemplo práctico de Ingeniería de Software: identificación de las clases en un documento de requerimientos

Ahora empezaremos a diseñar el sistema ATM que presentamos en el capítulo 2. En esta sección identificaremos las clases necesarias para crear el sistema ATM, analizando los sustantivos y las frases nominales que aparecen en el documento de requerimientos. Presentaremos los diagramas de clases de UML para modelar las relaciones entre estas clases. Este primer paso es importante para definir la estructura de nuestro sistema.

Identificación de las clases en un sistema

Para comenzar nuestro proceso de DOO, identificaremos las clases requeridas para crear el sistema ATM. Más adelante describiremos estas clases mediante el uso de los diagramas de clases de UML y las implementaremos en Java. Primero debemos revisar el documento de requerimientos de la sección 2.9, para identificar los sustantivos y frases nominales clave que nos ayuden a identificar las clases que conformarán el sistema ATM. Tal vez decidamos que algunos de estos sustantivos y frases nominales sean atributos de otras clases en el sistema. Tal vez también concluyamos que algunos de los sustantivos no corresponden a ciertas partes del sistema y, por ende, no deben modelarse. A medida que avancemos por el proceso de diseño podemos ir descubriendo clases adicionales.

La figura 3.19 lista los sustantivos y frases nominales que se encontraron en el documento de requerimientos de la sección 2.9. Los listaremos de izquierda a derecha, en el orden en el que los encontramos por primera vez en el documento de requerimientos. Sólo listaremos la forma singular de cada sustantivo o frase nominal.

Sustantivos y frases nominales en el documento de requerimientos del ATM		
banco	dinero / fondos	número de cuenta
ATM	pantalla	NIP
usuario	teclado numérico	base de datos del banco
cliente	dispensador de efectivo	solicitud de saldo
transacción	billete de $20 / efectivo	retiro
cuenta	ranura de depósito	depósito
saldo	sobre de depósito	

Figura 3.19 | Sustantivos y frases nominales en el documento de requerimientos del ATM.

Crearemos clases sólo para los sustantivos y frases nominales que tengan importancia en el sistema ATM. No modelamos “banco” como una clase, ya que el banco no es una parte del sistema ATM; el banco sólo quiere que nosotros construyamos el ATM. “Cliente” y “usuario” también representan entidades fuera del sistema; son importantes debido a que interactúan con nuestro sistema ATM, pero no necesitamos modelarlos como clases en el software del ATM. Recuerde que modelamos un usuario del ATM (es decir, un cliente del banco) como el actor en el diagrama de casos de uso de la figura 2.20.

No necesitamos modelar “billete de $20” ni “sobre de depósito” como clases. Éstos son objetos físicos en el mundo real, pero no forman parte de lo que se va a automatizar. Podemos representar en forma adecuada la presencia de billetes en el sistema, mediante el uso de un atributo de la clase que modela el dispensador de efectivo (en la sección 4.15 asignaremos atributos a las clases del sistema ATM). Por ejemplo, el dispensador de efectivo mantiene un conteo del número de billetes que contiene. El documento de requerimientos no dice nada acerca de lo que debe hacer el sistema con los sobres de depósito después de recibirlos. Podemos suponer que con sólo admitir la recepción de un sobre (una operación que realiza la clase que modela la ranura de depósito) es suficiente para representar la presencia de un sobre en el sistema (en la sección 6.14 asignaremos operaciones a las clases del sistema ATM).

En nuestro sistema ATM simplificado, lo más apropiado sería representar varios montos de “dinero”, incluyendo el “saldo” de una cuenta, como atributos de clases. De igual forma, los sustantivos “número de cuenta” y “NIP” representan piezas importantes de información en el sistema ATM. Son atributos importantes de una cuenta bancaria. Sin embargo, no exhiben comportamientos. Por ende, podemos modelarlos de la manera más apropiada como atributos de una clase de cuenta.

Aunque, con frecuencia, el documento de requerimientos describe una “transacción” en un sentido general, no modelaremos la amplia noción de una transacción financiera en este momento. En vez de ello, modelaremos los tres tipos de transacciones (es decir, “solicitud de saldo”, “retiro” y “depósito”) como clases individuales. Estas clases poseen los atributos específicos necesarios para ejecutar las transacciones que representan. Por ejemplo, para un retiro se necesita conocer el monto de dinero que el usuario desea retirar. Sin embargo, una solicitud de saldo no requiere datos adicionales. Lo que es más, las tres clases de transacciones exhiben comportamientos únicos. Para un retiro se requiere entregar efectivo al usuario, mientras que para un depósito se requiere recibir un sobre de depósito del usuario. [*Nota:* en la sección 10.9, “factorizaremos” las características comunes de todas las transacciones en una clase de “transacción” general, mediante el uso del concepto orientado a objetos de herencia].

Determinaremos las clases para nuestro sistema con base en los sustantivos y frases nominales restantes de la figura 3.19. Cada una de ellas se refiere a uno o varios de los siguientes elementos:

- ATM
- pantalla
- teclado numérico
- dispensador de efectivo
- ranura de depósito
- cuenta
- base de datos del banco
- solicitud de saldo
- retiro
- depósito

Es probable que los elementos de esta lista sean clases que necesitaremos implementar en nuestro sistema.

Ahora podemos modelar las clases en nuestro sistema, con base en la lista que hemos creado. En el proceso de diseño escribimos los nombres de las clases con la primera letra en mayúscula (una convención de UML), como lo haremos cuando escribamos el código de Java para implementar nuestro diseño. Si el nombre de una clase contiene más de una palabra, juntaremos todas las palabras y escribiremos la primera letra de cada una de ellas en mayúscula (por ejemplo, `NombreConVariasPalabras`). Utilizando esta convención, crearemos las clases `ATM`, `Pantalla`, `Teclado`, `DispensadorEfectivo`, `RanuraDeposito`, `Cuenta`, `BaseDatosBanco`, `SolicitudSaldo`, `Retiro` y `Deposito`. Construiremos nuestro sistema mediante el uso de todas estas clases como bloques de construcción. Sin embargo, antes de empezar a construir el sistema, debemos comprender mejor la forma en que las clases se relacionan entre sí.

Modelado de las clases

UML nos permite modelar, a través de los **diagramas de clases**, las clases en el sistema ATM y sus interrelaciones. La figura 3.20 representa a la clase `ATM`. En UML, cada clase se modela como un rectángulo con tres compartimientos. El compartimiento superior contiene el nombre de la clase, centrado horizontalmente y en negrita. El compartimiento intermedio contiene los atributos de la clase (en las secciones 4.15 y 5.11 hablaremos sobre los atributos). El compartimiento inferior contiene las operaciones de la clase (que veremos en la sección 6.14). En la figura 3.20, los compartimientos intermedio e inferior están vacíos, ya que no hemos determinado los atributos y operaciones de esta clase todavía.

Los diagramas de clases también muestran las relaciones entre las clases del sistema. La figura 3.21 muestra cómo nuestras clases `ATM` y `Retiro` se relacionan una con la otra. Por el momento modelaremos sólo este subconjunto de las clases del ATM, por cuestión de simpleza. Más adelante en esta sección, presentaremos un diagrama de clases más completo. Observe que los rectángulos que representan a las clases en este diagrama no están subdivididos en compartimientos. UML permite suprimir los atributos y las operaciones de una clase de esta forma, cuando sea apropiado, para crear diagramas más legibles. Un diagrama de este tipo se denomina **diagrama con elementos omitidos (elided diagram)**: su información, como el contenido de los compartimientos segundo y tercero, no se modela. En las secciones 4.15 y 6.14 colocaremos información en estos compartimientos.

En la figura 3.21, la línea sólida que conecta a las dos clases representa una **asociación**: una relación entre clases. Los números cerca de cada extremo de la línea son valores de **multiplicidad**; éstos indican cuántos objetos de cada clase participan en la asociación. En este caso, al seguir la línea de un extremo al otro se revela que, en un momento dado, un objeto `ATM` participa en una asociación con cero o con un objeto `Retiro`; cero si el usuario actual no está realizando una transacción o si ha solicitado un tipo distinto de transacción, y uno si el usuario ha solicitado un retiro. UML puede modelar muchos tipos de multiplicidad. La figura 3.22 lista y explica los tipos de multiplicidad.

Una asociación puede tener nombre. Por ejemplo, la palabra `Ejecuta` por encima de la línea que conecta a las clases `ATM` y `Retiro` en la figura 3.21 indica el nombre de esa asociación. Esta parte del diagrama se lee así: "un objeto de la clase `ATM` ejecuta cero o un objeto de la clase `Retiro`". Los nombres de las asociaciones son direccionales, como lo indica la punta de flecha rellena; por lo tanto, sería inapropiado, por ejemplo, leer la anterior asociación de derecha a izquierda como "cero o un objeto de la clase `Retiro` ejecuta un objeto de la clase `ATM`".

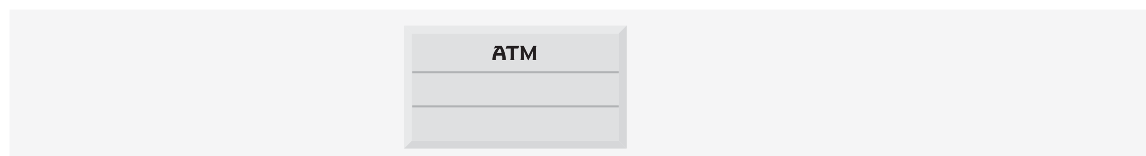


Figura 3.20 | Representación de una clase en UML mediante un diagrama de clases.



Figura 3.21 | Diagrama de clases que muestra una asociación entre clases.

Símbolo	Significado
0	Ninguno.
1	Uno.
m	Un valor entero.
0..1	Cero o uno.
m, n	*m* o *n*.
m..n	Cuando menos *m*, pero no más que *n*.
*	Cualquier entero no negativo (cero o más).
0..*	Cero o más (idéntico a *).
1..*	Uno o más.

Figura 3.22 | Tipos de multiplicidad.

La palabra `transaccionActual` en el extremo de `Retiro` de la línea de asociación en la figura 3.21 es un **nombre de rol**, el cual identifica el rol que desempeña el objeto `Retiro` en su relación con el `ATM`. Un nombre de rol agrega significado a una asociación entre clases, ya que identifica el rol que desempeña una clase dentro del contexto de una asociación. Una clase puede desempeñar varios roles en el mismo sistema. Por ejemplo, en un sistema de personal de una universidad, una persona puede desempeñar el rol de "profesor" con respecto a los estudiantes. La misma persona puede desempeñar el rol de "colega" cuando participa en una asociación con otro profesor, y de "entrenador" cuando entrena a los atletas estudiantes. En la figura 3.21, el nombre de rol `transaccionActual` indica que el objeto `Retiro` que participa en la asociación `Ejecuta` con un objeto de la clase `ATM` representa a la transacción que está procesando el ATM en ese momento. En otros contextos, un objeto `Retiro` puede desempeñar otros roles (por ejemplo, la transacción anterior). Observe que no especificamos un nombre de rol para el extremo del `ATM` de la asociación `Ejecuta`. A menudo, los nombres de los roles se omiten en los diagramas de clases, cuando el significado de una asociación está claro sin ellos.

Además de indicar relaciones simples, las asociaciones pueden especificar relaciones más complejas, como cuando los objetos de una clase están compuestos de objetos de otras clases. Considere un cajero automático real. ¿Qué "piezas" reúne un fabricante para construir un ATM funcional? Nuestro documento de requerimientos nos indica que el ATM está compuesto de una pantalla, un teclado, un dispensador de efectivo y una ranura de depósito.

En la figura 3.23, los **diamantes sólidos** que se adjuntan a las líneas de asociación de la clase `ATM` indican que esta clase tiene una relación de **composición** con las clases `Pantalla`, `Teclado`, `DispensadorEfectivo` y `RanuraDeposito`. La composición implica una relación todo/parte. La clase que tiene el símbolo de composición (el diamante sólido) en su extremo de la línea de asociación es el todo (en este caso, `ATM`), y las clases en el otro extremo de las líneas de asociación son las partes; en este caso, las clases `Pantalla`, `Teclado`, `DispensadorEfectivo` y `RanuraDeposito`. Las composiciones en la figura 3.23 indican que un objeto de la clase `ATM` está formado por un objeto de la clase `Pantalla`, un objeto de la clase `DispensadorEfectivo`, un objeto de la clase `Teclado` y un objeto de la clase `RanuraDeposito`. El ATM "tiene una" pantalla, un teclado, un dispensador de efectivo y una ranura de depósito. La **relación** *tiene un* define la composición (en la sección del Ejemplo práctico de Ingeniería de Software del capítulo 10 veremos que la relación "*es un*" define la herencia).

De acuerdo con la especificación del UML (`www.uml.org`), las relaciones de composición tienen las siguientes propiedades:

1. Sólo una clase en la relación puede representar el todo (es decir, el diamante puede colocarse sólo en un extremo de la línea de asociación). Por ejemplo, la pantalla es parte del ATM o el ATM es parte de la pantalla, pero la pantalla y el ATM no pueden representar ambos el "todo" dentro de la relación.
2. Las partes en la relación de composición existen sólo mientras exista el todo, y el todo es responsable de la creación y destrucción de sus partes. Por ejemplo, el acto de construir un ATM incluye la manufactura de sus partes. Lo que es más, si el ATM se destruye, también se destruyen su pantalla, teclado, dispensador de efectivo y ranura de depósito.
3. Una parte puede pertenecer sólo a un todo a la vez, aunque esa parte puede quitarse y unirse a otro todo, el cual entonces asumirá la responsabilidad de esa parte.

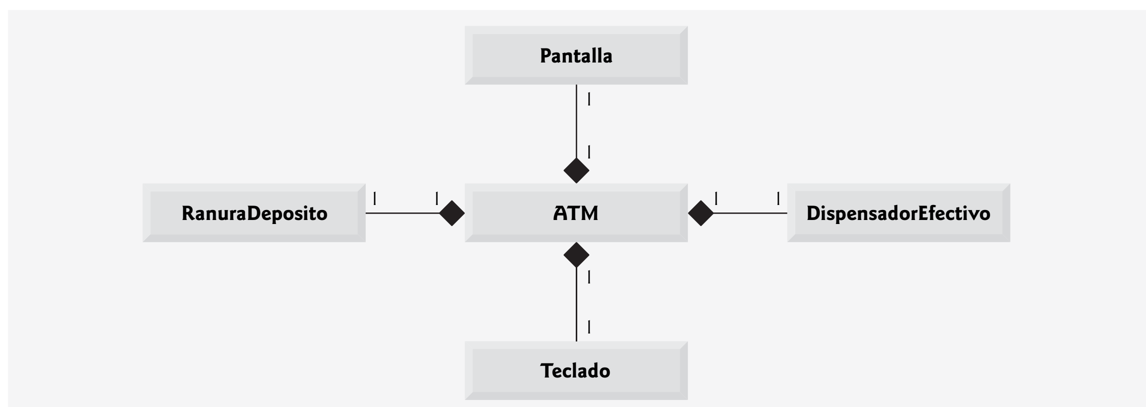


Figura 3.23 | Diagrama de clases que muestra las relaciones de composición.

Los diamantes sólidos en nuestros diagramas de clases indican las relaciones de composición que cumplen con estas tres propiedades. Si una relación "*es un*" no satisface uno o más de estos criterios, UML especifica que se deben adjuntar diamantes sin relleno a los extremos de las líneas de asociación para indicar una **agregación**: una forma más débil de la composición. Por ejemplo, una computadora personal y un monitor de computadora participan en una relación de agregación: la computadora "tiene un" monitor, pero las dos partes pueden existir en forma independiente, y el mismo monitor puede conectarse a varias computadoras a la vez, con lo cual se violan las propiedades segunda y tercera de la composición.

La figura 3.24 muestra un diagrama de clases para el sistema ATM. Este diagrama modela la mayoría de las clases que identificamos antes en esta sección, así como las asociaciones entre ellas que podemos inferir del documento de requerimientos. [*Nota:* las clases `SolicitudSaldo` y `Deposito` participan en asociaciones similares a las de la clase `Retiro`, por lo que preferimos omitirlas en este diagrama por cuestión de simpleza. En el capítulo 10 expandiremos nuestro diagrama de clases para incluir todas las clases en el sistema ATM].

La figura 3.24 presenta un modelo gráfico de la estructura del sistema ATM. Este diagrama de clases incluye a las clases `BaseDatosBanco` y `Cuenta`, junto con varias asociaciones que no presentamos en las figuras 3.21 o 3.23. El diagrama de clases muestra que la clase `ATM` tiene una **relación de uno a uno** con la clase `BaseDatosBanco`: un objeto `ATM` autentica a los usuarios en base a un objeto `BaseDatosBanco`. En la figura 3.24 también modelamos el hecho de que la base de datos del banco contiene información sobre muchas cuentas; un objeto de la clase `BaseDatosBanco` participa en una relación de composición con cero o más objetos de la clase `Cuenta`. Recuerde que en la figura 3.22 se muestra que el valor de multiplicidad `0..*` en el extremo de la clase `Cuenta`, de la asociación entre las clases `BaseDatosBanco` y `Cuenta`, indica que cero o más objetos de la clase `Cuenta` participan en la asociación. La clase `BaseDatosBanco` tiene una **relación de uno a varios** con la clase `Cuenta`; `BaseDatosBanco` puede contener muchos objetos `Cuenta`. De manera similar, la clase `Cuenta` tiene una **relación de varios a uno** con la clase `BaseDatosBanco`; puede haber muchos objetos `Cuenta` en `BaseDatosBanco`. [*Nota:* si recuerda la figura 3.22, el valor de multiplicidad * es idéntico a 0..*. Incluimos 0..* en nuestros diagramas de clases por cuestión de claridad].

La figura 3.24 también indica que si el usuario va a realizar un retiro, "un objeto de la clase `Retiro` accede a/modifica el saldo de una cuenta a través de un objeto de la clase `BaseDatosBanco`". Podríamos haber creado una asociación directamente entre la clase `Retiro` y la clase `Cuenta`. No obstante, el documento de requerimientos indica que el "ATM debe interactuar con la base de datos de información de las cuentas del banco" para realizar transacciones. Una cuenta de banco contiene información delicada, por lo que los ingenieros de sistemas deben considerar siempre la seguridad de los datos personales al diseñar un sistema. Por ello, sólo `BaseDatosBanco` puede acceder a una cuenta y manipularla en forma directa. Todas las demás partes del sistema deben interactuar con la base de datos para recuperar o actualizar la información de las cuentas (por ejemplo, el saldo de una cuenta).

El diagrama de clases de la figura 3.24 también modela las asociaciones entre la clase `Retiro` y las clases `Pantalla, DispensadorEfectivo` y `Teclado`. Una transacción de retiro implica pedir al usuario que seleccione el monto a retirar; también implica recibir entrada numérica. Estas acciones requieren el uso de la pantalla y del teclado, respectivamente. Además, para entregar efectivo al usuario se requiere acceso al dispensador de efectivo.

Aunque no se muestran en la figura 3.24, las clases `SolicitudSaldo` y `Deposito` participan en varias asociaciones con las otras clases del sistema ATM. Al igual que la clase `Retiro`, cada una de estas clases se asocia con las clases `ATM` y `BaseDatosBanco`. Un objeto de la clase `SolicitudSaldo` también se asocia con un objeto de la clase `Pantalla` para mostrar al usuario el saldo de una cuenta. La clase `Deposito` se asocia con las clases `Pantalla`, `Teclado` y `RanuraDeposito`. Al igual que los retiros, las transacciones de depósito requieren el uso de la pantalla y el teclado para mostrar mensajes y recibir datos de entrada, respectivamente. Para recibir sobres de depósito, un objeto de la clase `Deposito` accede a la ranura de depósitos.

Ya hemos identificado las clases en nuestro sistema ATM (aunque tal vez descubramos otras, a medida que avancemos con el diseño y la implementación). En la sección 4.15 determinaremos los atributos para cada una de estas clases, y en la sección 5.11 utilizaremos estos atributos para examinar la forma en que cambia el sistema con el tiempo.

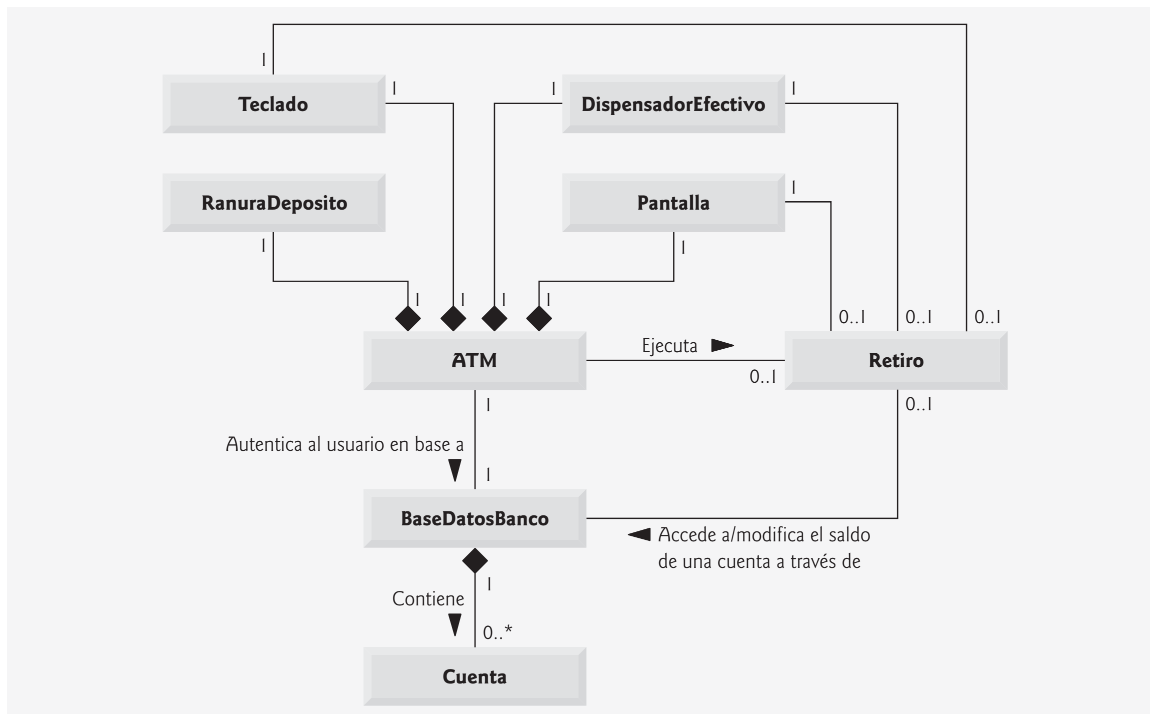


Figura 3.24 | Diagrama de clases para el modelo del sistema ATM.

Ejercicios de autoevaluación del Ejemplo práctico de Ingeniería de Software

3.1 Suponga que tenemos una clase llamada `Auto`, la cual representa a un automóvil. Piense en algunas de las distintas piezas que podría reunir un fabricante para producir un automóvil completo. Cree un diagrama de clases (similar a la figura 3.23) que modele algunas de las relaciones de composición de la clase `Auto`.

3.2 Suponga que tenemos una clase llamada `Archivo`, la cual representa un documento electrónico en una computadora independiente, sin conexión de red, representada por la clase `Computadora`. ¿Qué tipo de asociación existe entre la clase `Computadora` y la clase `Archivo`?

a) La clase `Computadora` tiene una relación de uno a uno con la clase `Archivo`.
b) La clase `Computadora` tiene una relación de varios a uno con la clase `Archivo`.
c) La clase `Computadora` tiene una relación de uno a varios con la clase `Archivo`.
d) La clase `Computadora` tiene una relación de varios a varios con la clase `Archivo`.

3.3 Indique si la siguiente aseveración es *verdadera* o *falsa*. Si es *falsa*, explique por qué: un diagrama de clases de UML, en el que no se modelan los compartimientos segundo y tercero, se denomina diagrama con elementos omitidos (elided diagram).

3.4 Modifique el diagrama de clases de la figura 3.24 para incluir la clase `Deposito`, en vez de la clase `Retiro`.

Respuestas a los ejercicios de autoevaluación del Ejemplo práctico de Ingeniería de Software

3.1 [*Nota:* las respuestas de los estudiantes pueden variar]. La figura 3.25 presenta un diagrama de clases que muestra algunas de las relaciones de composición de una clase `Auto`.

3.2 c. [*Nota:* en una computadora con conexión de red, esta relación podría ser de varios a varios].

3.3 Verdadera.

3.4 La figura 3.26 presenta un diagrama de clases para el ATM, en el cual se incluye la clase `Deposito` en vez de la clase `Retiro` (como en la figura 3.24). Observe que la clase `Deposito` no se asocia con la clase `DispensadorEfectivo`, sino que se asocia con la clase `RanuraDeposito`.

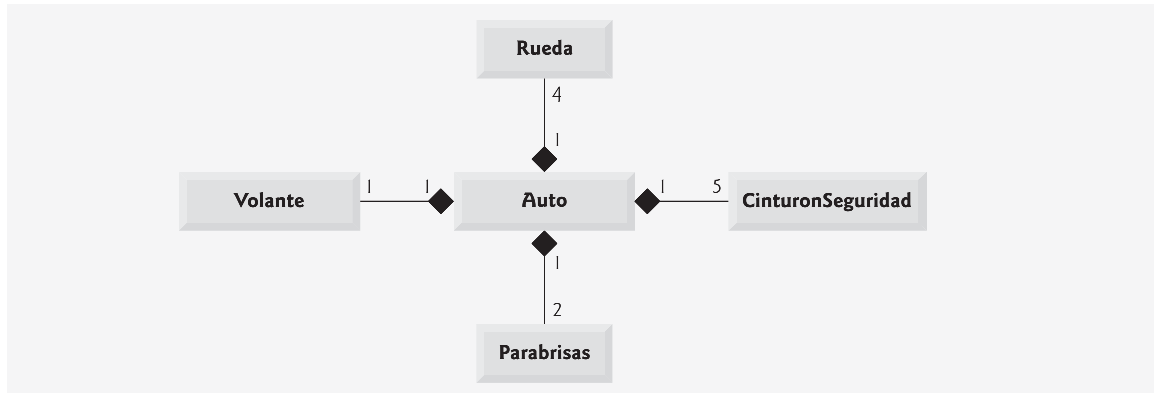


Figura 3.25 | Diagrama de clases que muestra algunas relaciones de composición de una clase `Auto`.

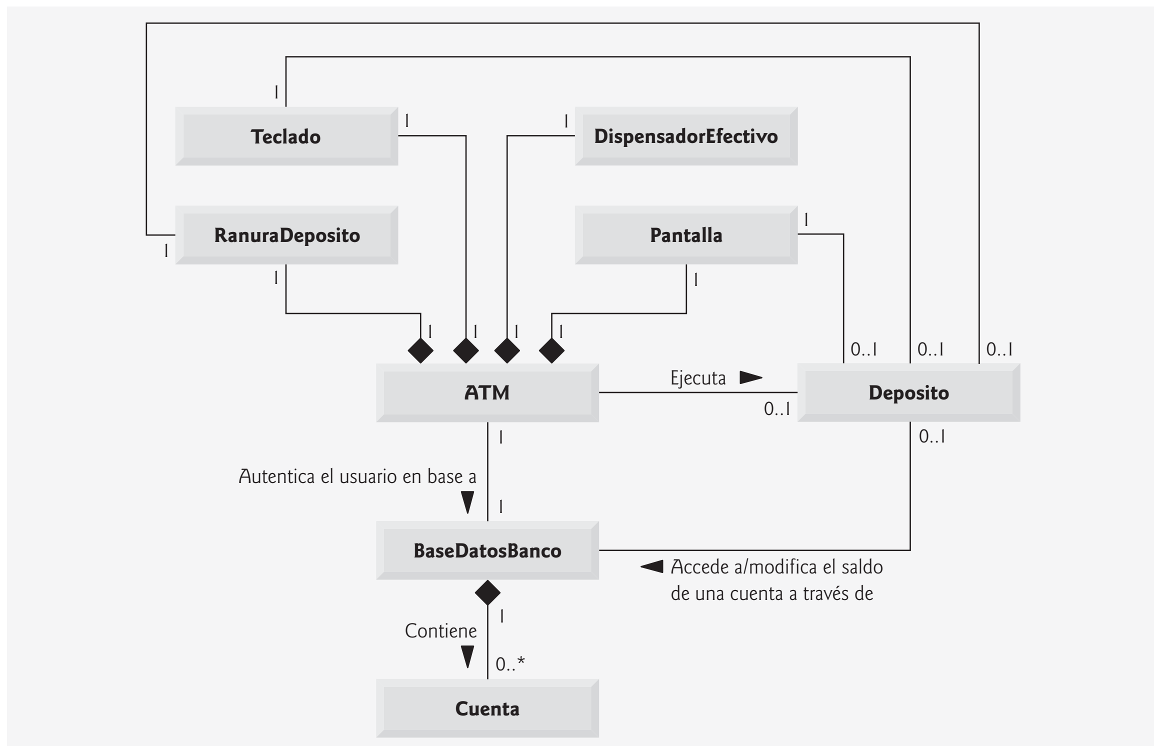


Figura 3.26 | Diagrama de clases para el modelo del sistema ATM, incluyendo la clase `Deposito`.

3.11 Conclusión

En este capítulo aprendió los conceptos básicos de las clases, los objetos, los métodos y las variables de instancia; utilizará estos conceptos en la mayoría de las aplicaciones de Java que vaya a crear. En especial, aprendió a declarar variables de instancia de una clase para mantener los datos de cada objeto de la clase, y cómo declarar métodos que operen sobre esos datos. Aprendió cómo llamar a un método para decirle que realice su tarea y cómo pasar información a los métodos en forma de argumentos. Vio la diferencia entre una variable local de un método y una variable de instancia de una clase, y que sólo las variables de instancia se inicializan en forma automática. También aprendió a utilizar el constructor de una clase para especificar los valores iniciales para las variables de instancia de un objeto. A lo largo del capítulo, vio cómo puede usarse UML para crear diagramas de clases que modelen los constructores, métodos y atributos de las clases. Por último, aprendió acerca de los números de punto flotante: cómo almacenarlos con variables del tipo primitivo `double`, cómo recibirlos en forma de datos de entrada mediante un objeto `Scanner` y cómo darles formato con `printf` y el especificador de formato `%f` para fines de visualización. En el siguiente capítulo empezaremos nuestra introducción a las instrucciones de control, las cuales especifican el orden en el que se realizan las acciones de un programa. Utilizará estas instrucciones en sus métodos para especificar cómo deben realizar sus tareas.

Resumen

Sección 3.2 Clases, objetos, métodos y variables de instancia

- Para realizar una tarea en un programa se requiere un método. Dentro del método se colocan los mecanismos que hacen que éste realice sus tareas; es decir, el método oculta los detalles de implementación de las tareas que realiza.
- La unidad de programa que aloja a un método se llama clase. Una clase puede contener uno o más métodos, que están diseñados para realizar las tareas de esa clase.
- Un método puede realizar una tarea y devolver un resultado.
- Puede utilizarse una clase para crear una instancia de la clase, a la cual se le llama objeto. Ésta es una de las razones por las que Java se conoce como lenguaje de programación orientado a objetos.
- Cada mensaje que se envía a un objeto se conoce como llamada a un método, y ésta le indica a un método del objeto que realice su tarea.
- Cada método puede especificar parámetros que representan la información adicional requerida por el método para realizar su tarea correctamente. La llamada a un método suministra valores (llamados argumentos) para los parámetros del método.
- Un objeto tiene atributos que se acarrean con el objeto, a medida que éste se utiliza en un programa. Estos atributos se especifican como parte de la clase del objeto. Los atributos se especifican en las clases mediante campos.

Sección 3.3 Declaración de una clase con un método, e instanciamiento de un objeto de una clase

- Cada declaración de clase que empieza con la palabra clave `public` debe almacenarse en un archivo que tenga exactamente el mismo nombre que la clase, y que termine con la extensión de nombre de archivo `.java`.
- La palabra clave `public` es un modificador de acceso.
- Cada declaración de clase contiene la palabra clave `class`, seguida inmediatamente por el nombre de la clase.
- La declaración de un método que empieza con la palabra clave `public` indica que el método está "disponible para el público"; es decir, lo pueden llamar otras clases declaradas fuera de la declaración de esa clase.
- La palabra clave `void` indica que un método realizará una tarea, pero no devolverá información cuando la termine.
- Por convención, los nombres de los métodos empiezan con la primera letra en minúscula, y todas las palabras subsiguientes en el nombre empiezan con la primera letra en mayúscula.
- Los paréntesis vacíos después del nombre de un método indican que éste no requiere parámetros para realizar su tarea.
- El cuerpo de todos los métodos está delimitado por llaves izquierda y derecha ({ y }).
- El cuerpo de un método contiene instrucciones que realizan la tarea de éste. Una vez que se ejecutan las instrucciones, el método ha terminado su tarea.
- Cuando intentamos ejecutar una clase, Java busca el método `main` de la clase para empezar la ejecución.
- Cualquier clase que contenga `public static void main( String args[] )` puede usarse para ejecutar una aplicación.
- Por lo general, no podemos llamar a un método que pertenece a otra clase, sino hasta crear un objeto de esa clase.

- Las expresiones de creación de instancias de clases que empiezan con la palabra clave `new` crean nuevos objetos.
- Para llamar a un método de un objeto, se pone después del nombre de la variable un separador punto (.), el nombre del método y un conjunto de paréntesis, que contienen los argumentos del método.
- En UML, cada clase se modela en un diagrama de clases en forma de rectángulo con tres compartimientos. El compartimiento superior contiene el nombre de la clase, centrado horizontalmente y en negrita. El compartimiento intermedio contiene los atributos de la clase, que corresponden a los campos en Java. El compartimiento inferior contiene las operaciones de la clase, que corresponden a los métodos y constructores en Java.
- Para modelar las operaciones, UML lista el nombre de la operación, seguido de un conjunto de paréntesis. Un signo más (+) enfrente del nombre de la operación indica que ésta es una operación `public` en UML (es decir, un método `public` en Java).

Sección 3.4 Declaración de un método con un parámetro

- A menudo, los métodos requieren información adicional para realizar sus tareas. Dicha información adicional se proporciona mediante argumentos en las llamadas a los métodos.
- El método `nextLine` de `Scanner` lee caracteres hasta encontrar una nueva línea, y después devuelve los caracteres que leyó en forma de un objeto `String`.
- El método `next` de `Scanner` lee caracteres hasta encontrar cualquier carácter de espacio en blanco, y después devuelve los caracteres que leyó en forma de un objeto `String`.
- Un método que requiere datos para realizar su tarea debe especificar esto en su declaración, para lo cual coloca información adicional en la lista de parámetros del método.
- Cada parámetro debe especificar tanto un tipo como un identificador.
- Cuando se hace la llamada a un método, sus argumentos se asignan a sus parámetros. Entonces, el cuerpo del método utiliza las variables de los parámetros para acceder a los valores de los argumentos.
- Un método puede especificar varios parámetros, separando un parámetro del otro mediante una coma.
- El número de argumentos en la llamada a un método debe coincidir con el número de parámetros en la lista de parámetros de la declaración del método. Además, los tipos de los argumentos en la llamada al método deben ser consistentes con los tipos de los parámetros correspondientes en la declaración del método.
- La clase `String` está en el paquete `java.lang` que, por lo general se importa de manera implícita en todos los archivos de código fuente.
- Hay una relación especial entre las clases que se compilan en el mismo directorio en el disco. De manera predeterminada, se considera que dichas clases están en el mismo paquete, al cual se le conoce como paquete predeterminado. Las clases en el mismo paquete se importan implícitamente en los archivos de código fuente de las otras clases que están en el mismo paquete. Por ende, no se requiere una declaración `import` cuando una clase en un paquete utiliza a otra clase en el mismo paquete.
- No se requiere una declaración `import` si siempre hacemos referencia a una clase con su nombre de clase completamente calificado.
- Para modelar un parámetro de una operación, UML lista el nombre del parámetro, seguido de dos puntos y el tipo del parámetro entre los paréntesis que van después del nombre de la operación.
- UML tiene sus propios tipos de datos, similares a los de Java. No todos los tipos de datos de UML tienen los mismos nombres que los tipos correspondientes en Java.
- El tipo `String` de UML corresponde al tipo `String` de Java.

Sección 3.5 Variables de instancia, métodos **establecer** *y métodos* **obtener**

- Las variables que se declaran en el cuerpo de un método específico se conocen como variables locales, y pueden utilizarse sólo en ese método.
- Por lo general, una clase consiste en uno o más métodos que manipulan los atributos (datos) pertenecientes a un objeto específico de esa clase. Los atributos se representan como campos en la declaración de una clase. Dichas variables se llaman campos, y se declaran dentro de la declaración de una clase, pero fuera de los cuerpos de las declaraciones de los métodos de esa clase.
- Cuando cada objeto de una clase mantiene su propia copia de un atributo, el campo que representa a ese atributo también se conoce como variable de instancia. Cada objeto (instancia) de la clase tiene una instancia separada de la variable en la memoria.
- La mayoría de las declaraciones de variables de instancia van precedidas por el modificador de acceso `private`. Las variables o métodos declarados con el modificador de acceso `private` sólo están accesibles para los métodos de la clase en la que están declarados.
- Al proceso de declarar variables de instancia con el modificador de acceso `private` se le conoce como ocultamiento de datos.

- Un beneficio de los campos es que todos los métodos de la clase pueden usarlos. Otra diferencia entre un campo y una variable local es que un campo tiene un valor inicial predeterminado, que Java proporciona cuando el programador no especifica el valor inicial del campo, pero una variable local no hace esto.
- El valor predeterminado para un campo de tipo `String` es `null`.
- Cuando se llama a un método que especifica un tipo de valor de retorno y completa su tarea, el método devuelve un resultado al método que lo llamó.
- A menudo, las clases proporcionan métodos `public` para permitir que los clientes de la clase *establezcan* u *obtengan* variables de instancia `private`. Los nombres de estos métodos no necesitan comenzar con *establecer* u *obtener*, pero esta convención de nomenclatura es muy recomendada en Java, y requerida para ciertos componentes de software de Java especiales, conocidos como JavaBeans.
- UML representa a las variables de instancia como atributos, listando el nombre del atributo, seguido de dos puntos y el tipo del atributo.
- En UML, los atributos privados van precedidos por un signo menos (-).
- Para indicar el tipo de valor de retorno de una operación, UML coloca dos puntos y el tipo de valor de retorno después de los paréntesis que siguen del nombre de la operación.
- Los diagramas de clases de UML no especifican tipos de valores de retorno para las operaciones que no devuelven valores.

Sección 3.6 Comparación entre tipos primitivos y tipos por referencia

- En Java, los tipos se dividen en dos categorías: tipos primitivos y tipos por referencia (algunas veces conocidos como tipos no primitivos). Los tipos primitivos son `boolean`, `byte`, `char`, `short`, `int`, `long`, `float` y `double`. Todos los demás tipos son por referencia, por lo cual, las clases que especifican los tipos de los objetos, son tipos por referencia.
- Una variable de tipo primitivo puede almacenar exactamente un valor de su tipo declarado, en un momento dado.
- Las variables de instancia de tipos primitivos se inicializan de manera predeterminada. Las variables de los tipos `byte`, `char`, `short`, `int`, `long`, `float` y `double` se inicializan con `0`. Las variables de tipo `boolean` se inicializan con `false`.
- Los programas utilizan variables de tipos por referencia (llamadas referencias) para almacenar la ubicación de un objeto en la memoria de la computadora. Dichas variables hacen referencia a los objetos en el programa. El objeto al que se hace referencia puede contener muchas variables de instancia y métodos.
- Los campos de tipo por referencia se inicializan de manera predeterminada con el valor `null`.
- Para invocar a los métodos de instancia de un objeto, se requiere una referencia a éste. Una variable de tipo primitivo no hace referencia a un objeto, por lo cual no puede usarse para invocar a un método.

Sección 3.7 Inicialización de objetos con constructores

- Un constructor puede usarse para inicializar un objeto de una clase, a la hora de crear este objeto.
- Los constructores pueden especificar parámetros, pero no tipos de valores de retorno.
- Si no se proporciona un constructor para una clase, el compilador proporciona un constructor predeterminado sin parámetros.
- Cuando una clase sólo tiene el constructor predeterminado, sus variables de instancia se inicializan con sus valores predeterminados. Las variables de los tipos `char`, `byte`, `short`, `int`, `long`, `float` y `double` se inicializan con `0`, las variables de tipo `boolean` se inicializan con `false`, y las variables de tipo por referencia se inicializan con `null`.
- Al igual que las operaciones, UML modela a los constructores en el tercer compartimiento de un diagrama de clases. Para diferenciar a un constructor en base a las operaciones de una clase, UML coloca la palabra “constructor” entre los signos « y » antes del nombre del constructor.

Sección 3.8 Números de punto flotante y el tipo **`double`**

- Un número de punto flotante es un número con un punto decimal, como 7.33, 0.0975 o 1000.12345. Java proporciona dos tipos primitivos para almacenar números de punto flotante en la memoria –`float` y `double`. La principal diferencia entre estos tipos es que las variables `double` pueden almacenar números con mayor magnitud y detalle (a esto se le conoce como la precisión del número) que las variables `float`.
- Las variables de tipo `float` representan números de punto flotante de precisión simple, y tienen siete dígitos significativos. Las variables de tipo `double` representan números de punto flotante de precisión doble. Éstos requieren el doble de memoria que las variables `float` y proporcionan 15 dígitos significativos; tienen aproximadamente el doble de precisión de las variables `float`.
- Los valores de punto flotante que aparecen en código fuente se conocen como literales de punto flotante, y son de tipo `double` de manera predeterminada.

- El método nextDouble de Scanner devuelve un valor double.
- El especificador de formato %f se utiliza para mostrar valores de tipo float o double. Puede especificarse una precisión entre % y f para representar el número de posiciones decimales que deben mostrarse a la derecha del punto decimal, en el número de punto flotante.
- El valor predeterminado para un campo de tipo double es 0.0, y el valor predeterminado para un campo de tipo int es 0.

Terminología

%f, especificador de formato
« y » (UML)
agregación (UML)
atributo (UML)
campo
campo de texto (GUI)
clase
class, palabra clave
cliente de un objeto de una clase
compartimiento en un diagrama de clases (UML)
componente de interfaz gráfica de usuario (GUI)
composición (UML)
constructor
constructor predeterminado
crear un objeto
cuadro de diálogo (GUI)
cuadro de diálogo de entrada (GUI)
cuadro de diálogo de mensaje (GUI)
declaración de clase
declarar un método
diagrama con elementos omitidos (UML)
diagrama de clases de UML
diálogo (GUI)
diamante sólido (UML)
double, tipo primitivo
encabezado de un método
enviar un mensaje
establecer, método
expresión de creación de instancia de clase
float, tipo primitivo
instancia de clase
instancia de una clase (objeto)
instanciar (o crear) un objeto
interfaz gráfica de usuario (GUI)
invocar a un método
JOptionPane, clase (GUI)
lenguaje extensible
lista de parámetros
literal de punto flotante
llamada a un método
mensaje
método
método que hace la llamada
modificador de acceso
multiplicidad (UML)
new, palabra clave
next, método de la clase Scanner
nextDouble, método de la clase Scanner
nextLine, método de la clase Scanner
nombre de rol (UML)
null, palabra reservada
número de punto flotante
número de punto flotante de precisión doble
número de punto flotante de precisión simple
objeto (o instancia)
ocultamiento de datos
operación (UML)
paquete predeterminado
parámetro
precisión de un número de punto flotante con formato
precisión de un valor de punto flotante
private, modificador de acceso
public, método
public, modificador de acceso
punto (.), separador
referencia
referirse a un objeto
relación "*tiene un*"
relación de uno a varios (UML)
relación de varios a uno (UML)
relación de varios a varios (UML)
showInputDialog, método de la clase JOptionPane (GUI)
showMessageDialog, método de la clase JOptionPane (GUI)
tipo de valor de retorno de un método
tipo por referencia
tipos no primitivos
valor inicial predeterminado
valor predeterminado
variable de instancia
variable local
void, palabra clave

Ejercicios de autoevaluación

3.1 Complete las siguientes oraciones:

a) Una casa es para un plano de construcción lo que un(a) ____________ para una clase.
b) Cada declaración de clase que empieza con la palabra clave ____________ debe almacenarse en un archivo que tenga exactamente el mismo nombre de la clase, y que termine con la extensión de nombre de archivo `.java`.
c) Cada declaración de clase contiene la palabra clave ____________, seguida inmediatamente por el nombre de la clase.
d) La palabra clave ____________ crea un objeto de la clase especificada a la derecha de la palabra clave.
e) Cada parámetro debe especificar un(a) ____________ y un(a) ____________.
f) De manera predeterminada, se considera que las clases que se compilan en el mismo directorio están en el mismo paquete, conocido como ____________.
g) Cuando cada objeto de una clase mantiene su propia copia de un atributo, el campo que representa a este atributo se conoce también como ____________.
h) Java proporciona dos tipos primitivos para almacenar números de punto flotante en la memoria: ____________ y ____________.
i) Las variables de tipo `double` representan a los números de punto flotante ____________.
j) El método ____________ de la clase `Scanner` devuelve un valor `double`.
k) La palabra clave `public` es un(a) ____________.
l) El tipo de valor de retorno ____________ indica que un método realizará una tarea, pero no devolverá información cuando complete su tarea.
m) El método ____________ de `Scanner` lee caracteres hasta encontrar una nueva línea, y después devuelve esos caracteres como un objeto `String`.
n) La clase `String` está en el paquete ____________.
o) No se requiere un(a) ____________ si siempre hacemos referencia a una clase con su nombre de clase completamente calificado.
p) Un ____________ es un número con un punto decimal, como 7.33, 0.0975 o 1000.12345.
q) Las variables de tipo `float` representan números de punto flotante ____________.
r) El especificador de formato ____________ se utiliza para mostrar valores de tipo `float` o `double`.
s) Los tipos en Java se dividen en dos categorías: tipos ____________ y tipos ____________.

3.2 Conteste con *verdadero* o *falso* a cada una de las siguientes proposiciones; en caso de ser *falso*, explique por qué.

a) Por convención, los nombres de los métodos empiezan con la primera letra en mayúscula y todas las palabras subsiguientes en el nombre empiezan con la primera letra en mayúscula.
b) Una declaración `import` no es obligatoria cuando una clase en un paquete utiliza a otra clase en el mismo paquete.
c) Los paréntesis vacíos que van después del nombre de un método en la declaración de un método indican que éste no requiere parámetros para realizar su tarea.
d) Las variables o los métodos declarados con el modificador de acceso `private` son accesibles sólo para los métodos de la clase en la que se declaran.
e) Una variable de tipo primitivo puede usarse para invocar un método.
f) Las variables que se declaran en el cuerpo de un método específico se conocen como variables de instancia, y pueden utilizarse en todos los métodos de la clase.
g) El cuerpo de cada método está delimitado por llaves izquierda y derecha ({ y }).
h) Las variables locales de tipo primitivo se inicializan de manera predeterminada.
i) Las variables de instancia de tipo por referencia se inicializan de manera predeterminada con el valor `null`.
j) Cualquier clase que contenga `public static void main( String args[] )` puede usarse para ejecutar una aplicación.
k) El número de argumentos en la llamada a un método debe coincidir con el número de parámetros en la lista de parámetros de la declaración del método.
l) Los valores de punto flotante que aparecen en código fuente se conocen como literales de punto flotante, y son de tipo `float` de manera predeterminada.

3.3 ¿Cuál es la diferencia entre una variable local y un campo?

3.4 Explique el propósito de un parámetro de un método. ¿Cuál es la diferencia entre un parámetro y un argumento?

Respuestas a los ejercicios de autoevaluación

3.1 a) objeto. b) `public`. c) `class`. d) `new`. e) tipo, nombre. f) paquete predeterminado. g) variable de instancia. h) `float`, `double`. i) de precisión doble. j) `nextDouble`. k) modificador de acceso. l) `void`. m) `nextLine`. n) `java.lang`. o) declaracion `import`. p) número de punto flotante. q) de precisión simple. r) `%f`. s) primitivo, por referencia.

3.2 a) Falso. Por convención, los nombres de los métodos empiezan con una primera letra en minúscula y todas las palabras subsiguientes en el nombre empiezan con una letra en mayúscula. b) Verdadero. c) Verdadero. d) Verdadero. e) Falso. Una variable de tipo primitivo no puede usarse para invocar a un método; se requiere una referencia a un objeto para invocar a los métodos de ese objeto. f) Falso. Dichas variables se llaman variables locales, y sólo se pueden utilizar en el método en el que están declaradas. g) Verdadero. h) Falso. Las variables de instancia de tipo primitivo se inicializan de manera predeterminada. A cada variable local se le debe asignar un valor de manera explícita. i) Verdadero. j) Verdadero. k) Verdadero. l) Falso. Dichas literales son de tipo `double` de manera predeterminada.

3.3 Una variable local se declara en el cuerpo de un método, y sólo puede utilizarse desde el punto en el que se declaró, hasta el final de la declaración del método. Un campo se declara en una clase, pero no en el cuerpo de alguno de los métodos de la clase. Cada objeto (instancia) de una clase tiene una copia separada de los campos de la clase. Además, los campos están accesibles para todos los métodos de la clase. (En el capítulo 8, Clases y objetos: un análisis más detallado, veremos una excepción a esto).

3.4 Un parámetro representa la información adicional que requiere un método para realizar su tarea. Cada parámetro requerido por un método está especificado en la declaración del método. Un argumento es el valor actual para un parámetro del método. Cuando se llama a un método, los valores de los argumentos se pasan al método, para que éste pueda realizar su tarea.

Ejercicios

3.5 ¿Cuál es el propósito de la palabra clave `new`? Explique lo que ocurre cuando se utiliza en una aplicación.

3.6 ¿Qué es un constructor predeterminado? ¿Cómo se inicializan las variables de instancia de un objeto, si una clase sólo tiene un constructor predeterminado?

3.7 Explique el propósito de una variable de instancia.

3.8 La mayoría de las clases necesitan importarse antes de poder utilizarlas en una aplicación ¿Por qué cualquier aplicación puede utilizar las clases `System` y `String` sin tener que importarlas primero?

3.9 Explique cómo utilizaría un programa la clase `Scanner`, sin importarla del paquete `java.util`.

3.10 Explique por qué una clase podría proporcionar un método *establecer* y un método *obtener* para una variable de instancia.

3.11 Modifique la clase `LibroCalificaciones` (figura 3.10) de la siguiente manera:

a) Incluya una segunda variable de instancia `String`, que represente el nombre del instructor del curso.
b) Proporcione un método *establecer* para modificar el nombre del instructor, y un método *obtener* para obtener el nombre.
c) Modifique el constructor para especificar dos parámetros: uno para el nombre del curso y otro para el nombre del instructor.
d) Modifique el método `mostrarMensaje`, de tal forma que primero imprima el mensaje de bienvenida y el nombre del curso, y que después imprima `"Este curso es presentado por: "`, seguido del nombre del instructor.

Use su clase modificada en una aplicación de prueba que demuestre las nuevas capacidades de la clase.

3.12 Modifique la clase `Cuenta` (figura 3.13) para proporcionar un método llamado `cargar`, que retire dinero de un objeto `Cuenta`. Asegure que el monto a cargar no exceda el saldo de `Cuenta`. Si lo hace, el saldo debe permanecer sin cambio y el método debe imprimir un mensaje que indique `"El monto a cargar excede el saldo de la cuenta"`. Modifique la clase `PruebaCuenta` (figura 3.14) para probar el método `cargar`.

3.13 Cree una clase llamada `Factura`, que una ferretería podría utilizar para representar una factura para un artículo vendido en la tienda. Una `Factura` debe incluir cuatro piezas de información como variables de instancia: un número de pieza (tipo `String`), la descripción de la pieza (tipo `String`), la cantidad de artículos de ese tipo que se van a comprar

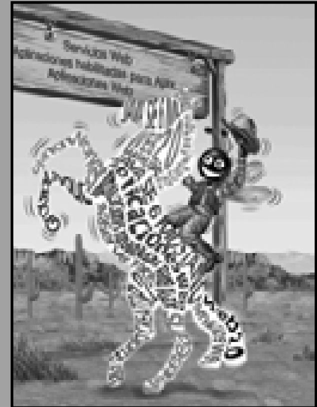
(tipo `int`) y el precio por artículo (`double`). Su clase debe tener un constructor que inicialice las cuatro variables de instancia. Proporcione un método *establecer* y un método *obtener* para cada variable de instancia. Además, proporcione un método llamado `obtenerMontoFactura`, que calcule el monto de la factura (es decir, que multiplique la cantidad por el precio por artículo) y después devuelva ese monto como un valor `double`. Si la cantidad no es positiva, debe establecerse en `0`. Si el precio por artículo no es positivo, debe establecerse a `0.0`. Escriba una aplicación de prueba llamada `PruebaFactura`, que demuestre las capacidades de la clase `Factura`.

3.14 Cree una clase llamada `Empleado`, que incluya tres piezas de información como variables de instancia: un primer nombre (tipo `String`), un apellido paterno (tipo `String`) y un salario mensual (`double`). Su clase debe tener un constructor que inicialice las tres variables de instancia. Proporcione un método *establecer* y un método *obtener* para cada variable de instancia. Si el salario mensual no es positivo, establézcalo a `0.0`. Escriba una aplicación de prueba llamada `PruebaEmpleado`, que demuestre las capacidades de cada `Empleado`. Cree dos objetos `Empleado` y muestre el salario *anual* de cada objeto. Después, proporcione a cada `Empleado` un aumento del 10% y muestre el salario anual de cada `Empleado` otra vez.

3.15 Cree una clase llamada `Fecha`, que incluya tres piezas de información como variables de instancia —un mes (tipo `int`), un día (tipo `int`) y un año (tipo `int`). Su clase debe tener un constructor que inicialice las tres variables de instancia, y debe asumir que los valores que se proporcionan son correctos. Proporcione un método *establecer* y un método *obtener* para cada variable de instancia. Proporcione un método `mostrarFecha`, que muestre el mes, día y año, separados por barras diagonales (`/`). Escriba una aplicación de prueba llamada `PruebaFecha`, que demuestre las capacidades de la clase `Fecha`.

4

Instrucciones de control: parte 1



Desplacémonos un lugar.

—Lewis Carroll

La rueda se convirtió en un círculo completo.

—William Shakespeare

¡Cuántas manzanas tuvieron que caer en la cabeza de Newton antes de que entendiera el suceso!

—Robert Frost

Toda la evolución que conocemos procede de lo vago a lo definido.

—Charles Sanders Peirce

OBJETIVOS

En este capítulo aprenderá a:

- Comprender las técnicas básicas para solucionar problemas.
- Desarrollar algoritmos mediante el proceso de refinamiento de arriba a abajo, paso a paso, usando seudocódigo.
- Utilizar las estructuras de selección `if` e `if...else` para elegir entre distintas acciones alternativas.
- Utilizar la estructura de repetición `while` para ejecutar instrucciones de manera repetitiva dentro de un programa.
- Comprender la repetición controlada por un contador y la repetición controlada por un centinela.
- Utilizar los operadores de asignación compuestos, de incremento y decremento.
- Conocer los tipos de datos primitivos.

Plan general

4.1 Introducción

Antes de escribir un programa que dé solución a un problema, es imprescindible tener una comprensión detallada de todo el problema, además de una metodología cuidadosamente planeada para resolverlo. Al escribir un programa, es igualmente esencial comprender los tipos de bloques de construcción disponibles, y emplear las técnicas comprobadas para construir programas. En este capítulo y en el 5, Instrucciones de control: parte 2, hablaremos sobre estas cuestiones cuando presentemos la teoría y los principios de la programación estructurada. Los conceptos aquí presentados son imprescindibles para crear clases y manipular objetos.

En este capítulo presentamos las instrucciones `if...else` y `while` de Java, tres de los bloques de construcción que permiten a los programadores especificar la lógica requerida para que los métodos realicen sus tareas. Dedicamos una parte de este capítulo (y de los capítulos 5 y 7) para desarrollar más la clase `LibroCalificaciones` que presentamos en el capítulo 3. En especial, agregamos un método a la clase `LibroCalificaciones` que utiliza instrucciones de control para calcular el promedio de un conjunto de calificaciones de estudiantes. Otro ejemplo demuestra formas adicionales de combinar instrucciones de control para resolver un problema similar. Presentamos los operadores de asignación compuestos de Java, y exploramos los operadores de incremento y decremento. Estos operadores adicionales abrevian y simplifican muchas instrucciones de los programas. Por último, presentamos las generalidades acerca de los tipos de datos primitivos que están disponibles para los programadores.

4.2 Algoritmos

Cualquier problema de computación puede resolverse ejecutando una serie de acciones en un orden específico. Un procedimiento para resolver un problema en términos de:

1. las **acciones** a ejecutar y
2. el **orden** en el que se ejecutan estas acciones

se conoce como un **algoritmo**. El siguiente ejemplo demuestra que es importante especificar de manera correcta el orden en el que se ejecutan las acciones.

Considere el "algoritmo para levantarse y arreglarse" que sigue un ejecutivo para levantarse de la cama e ir a trabajar: (1) levantarse; (2) quitarse la pijama; (3) bañarse; (4) vestirse; (5) desayunar; (6) transportarse al trabajo. Esta rutina logra que el ejecutivo llegue al trabajo bien preparado para tomar decisiones críticas. Suponga

que los mismos pasos se realizan en un orden ligeramente distinto: (1) levantarse; (2) quitarse la pijama; (3) vestirse; (4) bañarse; (5) desayunar; (6) transportarse al trabajo. En este caso nuestro ejecutivo llegará al trabajo todo mojado.

Al proceso de especificar el orden en el que se ejecutan las instrucciones (acciones) en un programa, se le llama **control del programa**. En este capítulo investigaremos el control de los programas mediante el uso de las **instrucciones de control** de Java.

4.3 Seudocódigo

El **seudocódigo** es un lenguaje informal que ayuda a los programadores a desarrollar algoritmos sin tener que preocuparse por los estrictos detalles de la sintaxis del lenguaje Java. El seudocódigo que presentaremos es especialmente útil para desarrollar algoritmos que se convertirán en porciones estructuradas de programas en Java. El seudocódigo es similar al lenguaje cotidiano; es conveniente y amigable con el usuario, aunque no es realmente un lenguaje de programación de computadoras. Empezaremos a utilizar el seudocódigo en la sección 4.5, y en la figura 4.5 aparece un programa de seudocódigo de ejemplo.

El seudocódigo no se ejecuta en las computadoras. En vez de ello, ayuda al programador a "organizar" un programa antes de que intente escribirlo en un lenguaje de programación como Java. Este capítulo presenta varios ejemplos de cómo utilizar el seudocódigo para desarrollar programas en Java.

El estilo de seudocódigo que presentaremos consiste solamente en caracteres, de manera que los programadores pueden escribir el seudocódigo, utilizando cualquier programa editor de texto. Un programa en seudocódigo preparado de manera cuidadosa puede convertirse fácilmente en su correspondiente programa en Java. En muchos casos, esto requiere tan sólo reemplazar las instrucciones en seudocódigo con sus instrucciones equivalentes en Java.

Por lo general, el seudocódigo describe sólo las instrucciones que representan las acciones que ocurren después de que un programador convierte un programa de seudocódigo a Java, y el programa se ejecuta en una computadora. Dichas acciones podrían incluir la entrada, salida o un cálculo. Por lo general no incluimos las declaraciones de variables en nuestro seudocódigo, pero algunos programadores optan por listar las variables y mencionar sus propósitos al principio de su seudocódigo.

4.4 Estructuras de control

Generalmente, en un programa las instrucciones se ejecutan una después de otra, en el orden en que están escritas. Este proceso se conoce como **ejecución secuencial**. Varias instrucciones en Java, que pronto veremos, permiten al programador especificar que la siguiente instrucción a ejecutarse tal vez no sea la siguiente en la secuencia. Esto se conoce como **transferencia de control**.

Durante la década de los sesenta, se hizo evidente que el uso indiscriminado de las transferencias de control era el origen de muchas de las dificultades que experimentaban los grupos de desarrollo de software. A quien se señaló como culpable fue a la **instrucción `goto`** (utilizada en la mayoría de los lenguajes de programación de esa época), la cual permite al programador especificar la transferencia de control a uno de los muchos posibles destinos dentro de un programa. La noción de la llamada **programación estructurada** se hizo casi un sinónimo de la "eliminación del `goto`". [*Nota:* Java no tiene una instrucción `goto`; sin embargo, la palabra `goto` está reservada para Java y no debe usarse como identificador en los programas].

Las investigaciones de Bohm y Jacopini[1] demostraron que los programas podían escribirse sin instrucciones `goto`. El reto de la época para los programadores fue cambiar sus estilos a una "programación sin `goto`". No fue sino hasta la década de los setenta cuando los programadores tomaron en serio la programación estructurada. Los resultados fueron impresionantes. Los grupos de desarrollo de software reportaron reducciones en los tiempos de desarrollo, mayor incidencia de entregas de sistemas a tiempo y más proyectos de software finalizados sin salirse del presupuesto. La clave para estos logros fue que los programas estructurados eran más claros, más fáciles de depurar y modificar, y había más probabilidad de que estuvieran libres de errores desde el principio.

---

1. Bohm, C. y G. Jacopini, "Flow Diagrams, Turing Machines and Languages with Only Two Formation Rules", *Communications of the ACM*, vol. 9, núm. 5, mayo de 1966, páginas 336-371.

El trabajo de Bohm y Jacopini demostró que todos los programas podían escribirse en términos de tres estructuras de control solamente: la **estructura de secuencia**, la **estructura de selección** y la **estructura de repetición**. El término "estructuras de control" proviene del campo de las ciencias computacionales. Cuando presentemos las implementaciones de las estructuras de control en Java, nos referiremos a ellas en la terminología de la *Especificación del lenguaje Java* como "instrucciones de control".

Estructura de secuencia en Java

La estructura de secuencia está integrada en Java. A menos que se le indique lo contrario, la computadora ejecuta las instrucciones en Java una después de otra, en el orden en que estén escritas; es decir, en secuencia. El **diagrama de actividad** de la figura 4.1 ilustra una estructura de secuencia típica, en la que se realizan dos cálculos en orden. Java permite tantas acciones como deseemos en una estructura de secuencia. Como veremos pronto, en donde quiera que se coloque una sola acción, podrán colocarse varias acciones en secuencia.

Los diagramas de actividad son parte de UML. Un diagrama de actividad modela el **flujo de trabajo** (también conocido como la **actividad**) de una parte de un sistema de software. Dichos flujos de trabajo pueden incluir una porción de un algoritmo, como la estructura de secuencia de la figura 4.1. Los diagramas de actividad están compuestos por símbolos de propósito especial, como los **símbolos de estado de acción** (rectángulos cuyos lados izquierdo y derecho se reemplazan con arcos hacia fuera), **rombos** (**diamantes**) y **pequeños círculos**. Estos símbolos se conectan mediante **flechas de transición**, que representan el flujo de la actividad; es decir, el orden en el que deben ocurrir las acciones.

Al igual que el seudocódigo, los diagramas de actividad ayudan a los programadores a desarrollar y representar algoritmos; sin embargo, muchos de ellos aún prefieren el seudocódigo. Los diagramas de actividad muestran claramente cómo operan las estructuras de control.

Considere el diagrama de actividad para la estructura de secuencia de la figura 4.1. Este diagrama contiene dos **estados de acción** que representan las acciones a realizar. Cada estado de acción contiene una **expresión de acción** (por ejemplo, "sumar calificación a total" o "sumar 1 al contador"), que especifica una acción particular a realizar. Otras acciones podrían incluir cálculos u operaciones de entrada/salida. Las flechas en el diagrama de actividad representan **transiciones**, las cuales indican el orden en el que ocurren las acciones representadas por los estados de acción. El programa que implementa las actividades ilustradas por el diagrama de la figura 4.1 primero suma `calificacion` a `total`, y después suma `1` a `contador`.

El **círculo relleno** que se encuentra en la parte superior del diagrama de actividad representa el **estado inicial** de la actividad: el inicio del flujo de trabajo antes de que el programa realice las actividades modeladas. El **círculo sólido rodeado por una circunferencia** que aparece en la parte inferior del diagrama representa el **estado final**; es decir, el final del flujo de trabajo después de que el programa realiza sus acciones.

La figura 4.1 también incluye rectángulos que tienen la esquina superior derecha doblada. En UML, a estos rectángulos se les llama **notas** (como los comentarios en Java): comentarios con explicaciones que describen el propósito de los símbolos en el diagrama. La figura 4.1 utiliza las notas de UML para mostrar el código en Java asociado con cada uno de los estados de acción en el diagrama de actividad. Una **línea punteada** conecta cada nota con el elemento que ésta describe. Los diagramas de actividad generalmente no muestran el código en Java que implementa la actividad. En este libro utilizamos las notas con este propósito, para mostrar cómo se rela-

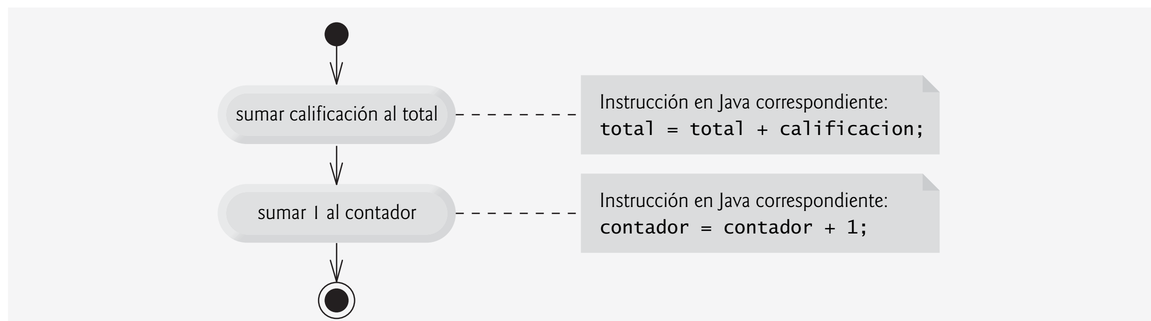


Figura 4.1 | Diagrama de actividad de una estructura de secuencia.

ciona el diagrama con el código en Java. Para obtener más información sobre UML, vea nuestro ejemplo práctico opcional, que aparece en las secciones tituladas Ejemplo práctico de Ingeniería de Software al final de los capítulos 1 al 8 y 10, o visite `www.uml.org`.

Instrucciones de selección en Java

Java tiene tres tipos de **instrucciones de selección** (las cuales se describen en este capítulo y en el siguiente). La instrucción `if` realiza (selecciona) una acción si la condición es verdadera, o evita la acción si la condición es falsa. La instrucción `if...else` realiza una acción si la condición es verdadera, o realiza una acción distinta si la condición es falsa. La instrucción `switch` (capítulo 5) realiza una de entre varias acciones distintas, dependiendo del valor de una expresión.

La instrucción `if` es una **instrucción de selección simple**, ya que selecciona o ignora una sola acción (o, como pronto veremos, un solo grupo de acciones). La instrucción `if...else` se conoce como **instrucción de selección doble**, ya que selecciona entre dos acciones distintas (o grupos de acciones). La instrucción `switch` es una **estructura de selección múltiple**, ya que selecciona entre diversas acciones (o grupos de acciones).

Instrucciones de repetición en Java

Java cuenta con tres instrucciones de repetición (también llamadas **instrucciones de ciclo**) que permiten a los programas ejecutar instrucciones en forma repetida, siempre y cuando una condición (llamada la **condición de continuación del ciclo**) siga siendo verdadera. Las instrucciones de repetición se implementan con las instrucciones `while`, `do...while` y `for`. (El capítulo 5 presenta las instrucciones `do...while` y `for`). Las instrucciones `while` y `for` realizan la acción (o grupo de acciones) en sus cuerpos, cero o más veces; si la condición de continuación del ciclo es inicialmente falsa, no se ejecutará la acción (o grupo de acciones). La instrucción `do...while` realiza la acción (o grupo de acciones) en su cuerpo, una o más veces.

Las palabras `if`, `else`, `switch`, `while`, `do` y `for` son palabras clave en Java; se utilizan para implementar varias características de Java, como las instrucciones de control. Las palabras clave no pueden usarse como identificadores, como los nombres de variables. En el apéndice C aparece una lista completa de las palabras clave en Java.

Resumen de las instrucciones de control en Java

Java sólo tiene tres tipos de estructuras de control, a las cuales nos referiremos de aquí en adelante como instrucciones de control: la instrucción de secuencia, las instrucciones de selección (tres tipos) y las instrucciones de repetición (tres tipos). Cada programa se forma combinando tantas instrucciones de secuencia, selección y repetición como sea apropiado para el algoritmo que implemente el programa. Al igual que con la instrucción de secuencia de la figura 4.1, podemos modelar cada una de las instrucciones de control como un diagrama de actividad. Cada diagrama contiene un estado inicial y final, los cuales representan el punto de entrada y salida de la instrucción de control, respectivamente. Las **instrucciones de control de una sola entrada/una sola salida** facilitan la creación de programas; las instrucciones de control están "unidas" entre sí mediante la conexión del punto de salida de una instrucción de control, al punto de entrada de la siguiente. Este procedimiento es similar a la manera en que un niño apila los bloques de construcción, así que a esto le llamamos **apilamiento de instrucciones de control**. En breve aprenderemos que sólo hay una manera alternativa de conectar las instrucciones de control: el **anidamiento de instrucciones de control**, en el cual una instrucción de control aparece dentro de otra. Por lo tanto, los algoritmos en los programas en Java se crean a partir de sólo tres principales tipos de instrucciones de control, que se combinan sólo de dos formas. Ésta es la esencia de la simpleza.

4.5 Instrucción de selección simple `if`

Los programas utilizan instrucciones de selección para elegir entre los cursos alternativos de acción. Por ejemplo, suponga que la calificación para aprobar un examen es 60. La instrucción en seudocódigo

Si la calificación del estudiante es mayor o igual a 60
Imprimir "Aprobado"

determina si la condición "la calificación del estudiante es mayor o igual a 60" es verdadera o falsa. Si la condición es verdadera se imprime "Aprobado", y se "ejecuta" en orden la siguiente instrucción en seudocódigo. (Recuerde que el seudocódigo no es un verdadero lenguaje de programación). Si la condición es falsa se ignora la instrucción

Imprimir, y se ejecuta en orden la siguiente instrucción en seudocódigo. La sangría de la segunda línea de esta instrucción de selección es opcional, pero se recomienda ya que enfatiza la estructura inherente de los programas estructurados.

La instrucción anterior *if* en seudocódigo puede escribirse en Java de la siguiente manera:

```
if ( calificacionEstudiante >= 60 )
   System.out.println( "Aprobado" );
```

Observe que el código en Java corresponde en gran medida con el seudocódigo. Ésta es una de las propiedades que hace del seudocódigo una herramienta de desarrollo de programas tan útil.

La figura 4.2 muestra la instrucción `if` de selección simple. Esta figura contiene lo que quizá sea el símbolo más importante en un diagrama de actividad: el rombo o **símbolo de decisión**, el cual indica que se tomará una decisión. El flujo de trabajo continuará a lo largo de una ruta determinada por las **condiciones de guardia** asociadas de ese símbolo, que pueden ser verdaderas o falsas. Cada flecha de transición que sale de un símbolo de decisión tiene una condición de guardia (especificada entre corchetes, a un lado de la flecha de transición). Si una condición de guardia es verdadera, el flujo de trabajo entra al estado de acción al que apunta la flecha de transición. En la figura 4.2, si la calificación es mayor o igual a 60, el programa imprime "Aprobado" y luego se dirige al estado final de esta actividad. Si la calificación es menor a 60, el programa se dirige inmediatamente al estado final sin mostrar ningún mensaje.

La instrucción `if` es una instrucción de control de una sola entrada/una sola salida. Pronto veremos que los diagramas de actividad para las instrucciones de control restantes también contienen estados iniciales, flechas de transición, estados de acción que indican las acciones a realizar, símbolos de decisión (con sus condiciones de guardia asociadas) que indican las decisiones a tomar, y estados finales. Esto es consistente con el **modelo de programación acción/decisión** que hemos estado enfatizando.

Imagine siete cajones, en donde cada uno contiene sólo un tipo de instrucción de control de Java. Todas las instrucciones de control están vacías. Su tarea es ensamblar un programa a partir de tantas instrucciones de control de cada tipo como lo requiera el algoritmo, combinando esas instrucciones de control en sólo dos formas posibles (apilando o anidando), y después llenando los estados de acción y las decisiones con expresiones de acción y condiciones de guardia, en una manera que sea apropiada para el algoritmo. Hablaremos sobre la variedad de formas en que pueden escribirse las acciones y las decisiones.

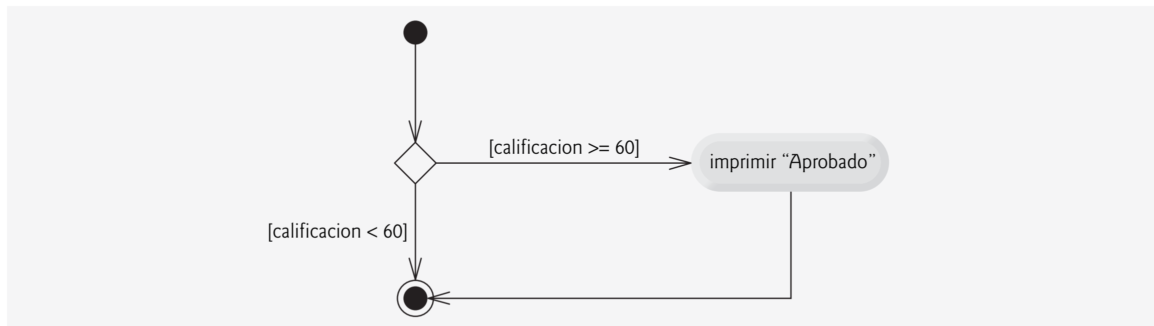


Figura 4.2 | Diagrama de actividad en UML de la instrucción `if` de selección simple.

4.6 Instrucción de selección doble `if...else`

La instrucción `if` de selección simple realiza una acción indicada solamente cuando la condición es verdadera (`true`); de no ser así, se evita dicha acción. La instrucción `if...else` de selección doble permite al programador especificar una acción a realizar cuando la condición es verdadera, y otra distinta cuando la condición es falsa. Por ejemplo, la instrucción en seudocódigo:

Si la calificación del estudiante es mayor o igual a 60
Imprimir "Aprobado"
De lo contrario
Imprimir "Reprobado"

imprime "Aprobado" si la calificación del estudiante es mayor o igual a 60, y, "Reprobado" si la calificación del estudiante es menor a 60. En cualquier caso, después de que ocurre la impresión se "ejecuta", según la secuencia, la siguiente instrucción en seudocódigo.

La instrucción anterior *if...else* en seudocódigo puede escribirse en Java como

```
if ( calificacion >= 60 )
   System.out.println( "Aprobado" );
else
   System.out.println( "Reprobado" );
```

Observe que el cuerpo de la instrucción `else` también tiene sangría. Cualquiera que sea la convención de sangría que usted elija, debe aplicarla consistentemente en todos sus programas. Es difícil leer programas que no obedecen las convenciones de espaciado uniformes.



Buena práctica de programación 4.1

Utilice sangría en ambos cuerpos de instrucciones de una estructura `if...else`.



Buena práctica de programación 4.2

Si hay varios niveles de sangría, en cada nivel debe aplicarse la misma cantidad de espacio adicional.

La figura 4.3 muestra el flujo de control en la instrucción `if...else`. Una vez más (además del estado inicial, las flechas de transición y el estado final), los símbolos en el diagrama de actividad de UML representan estados de acción y decisiones. Nosotros seguimos enfatizando este modelo de computación acción/decisión. Imagine de nuevo un cajón profundo que contiene tantas instrucciones `if...else` vacías como sea necesario para crear cualquier programa en Java. Su trabajo es ensamblar estas instrucciones `if...else` (apilando o anidando) con cualquier otra estructura de control requerida por el algoritmo. Usted debe llenar los estados de acción y los símbolos de decisión con expresiones de acción y condiciones de guardia que sean apropiadas para el algoritmo que esté desarrollando.

Operador condicional (`?:`)

Java cuenta con el **operador condicional** (`?:`), que en ocasiones puede utilizarse en lugar de una instrucción `if...else`. Éste es el único **operador ternario** en Java; es decir, que utiliza tres operandos. En conjunto, los operandos y el símbolo `?:` forman una **expresión condicional**. El primer operando (a la izquierda del `?`) es una expresión **`booleana`** (es decir, una condición que se evalúa a un valor `booleano`: **`true`** o **`false`**), el segundo operando (entre el `?` y `:`) es el valor de la expresión condicional si la expresión `booleana` es verdadera, y el tercer operando (a la derecha de `:`) es el valor de la expresión condicional si la expresión `booleana` se evalúa como `false`. Por ejemplo, la instrucción

```
System.out.println( calificacionEstudiante >= 60 ? "Aprobado" : "Reprobado" );
```

imprime el valor del argumento de `println`, que es una expresión condicional. La expresión condicional en esta instrucción produce como resultado la cadena `"Aprobado"` si la expresión `booleana calificacionEstudiante >= 60` es verdadera, o produce como resultado la cadena `"Reprobado"` si la expresión `booleana` es falsa. Por lo tanto, esta instrucción con el operador condicional realiza en esencia la misma función que la instrucción `if...else` que se mostró anteriormente, en esta sección. La precedencia del operador condicional es baja, por lo que toda la expresión condicional se coloca normalmente entre paréntesis. Pronto veremos que las expresiones condicionales pueden usarse en algunas situaciones en las que no se pueden utilizar instrucciones `if...else`.



Buena práctica de programación 4.3

Las expresiones condicionales son más difíciles de leer que las instrucciones `if...else`, por lo cual deben usarse para reemplazar sólo a las instrucciones `if...else` simples que seleccionan uno de dos valores.

Instrucciones `if...else` anidadas

Un programa puede evaluar varios casos colocando instrucciones `if...else` dentro de otras instrucciones `if...else`, para crear **instrucciones `if...else` anidadas**. Por ejemplo, el siguiente seudocódigo representa una ins-

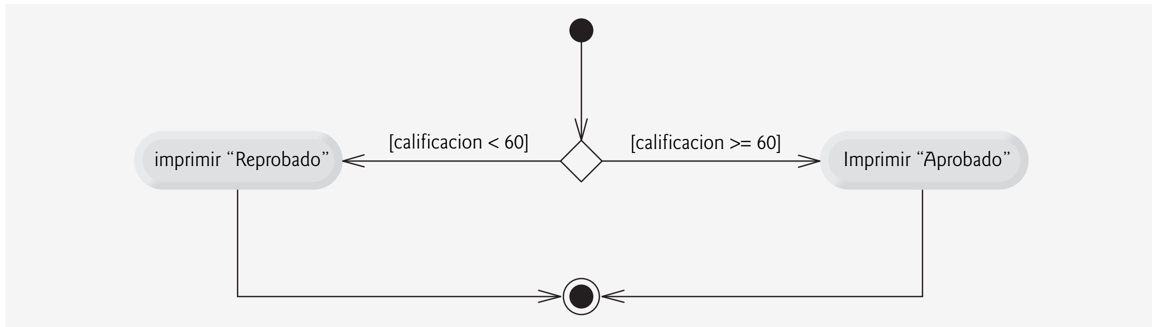


Figura 4.3 | Diagrama de actividad de UML de la instrucción `if...else` de selección doble.

trucción `if...else` anidada que imprime `A` para las calificaciones de exámenes mayores o iguales a 90, `B` para las calificaciones en el rango de 80 a 89, `C` para las calificaciones en el rango de 70 a 79, `D` para las calificaciones en el rango de 60 a 69 y `F` para todas las demás calificaciones:

Si la calificación del estudiante es mayor o igual a 90
    Imprimir "A"
de lo contrario
    Si la calificación del estudiante es mayor o igual a 80
        Imprimir "B"
    de lo contrario
        Si la calificación del estudiante es mayor o igual a 70
            Imprimir "C"
        de lo contrario
            Si la calificación del estudiante es mayor o igual a 60
                Imprimir "D"
            de lo contrario
                Imprimir "F"

Este seudocódigo puede escribirse en Java como

```
if ( calificacionEstudiante >= 90 )
   System.out.println( "A" );
else
   if ( calificacionEstudiante >= 80 )
      System.out.println( "B" );
   else
      if ( calificacionEstudiante >= 70 )
         System.out.println( "C" );
      else
         if ( calificacionEstudiante >= 60 )
            System.out.println( "D" );
         else
            System.out.println( "F" );
```

Si `calificacionEstudiante` es mayor o igual a 90, las primeras cuatro condiciones serán verdaderas, pero sólo se ejecutará la instrucción en la parte `if` de la primera instrucción `if...else`. Después de que se ejecute esa instrucción, se evita la parte `else` de la instrucción `if...else` más "externa". La mayoría de los programadores en Java prefieren escribir la instrucción `if...else` anterior así:

```
if ( calificacionEstudiante >= 90 )
   System.out.println( "A" );
else if ( calificacionEstudiante >= 80 )
   System.out.println( "B" );
else if ( calificacionEstudiante >= 70 )
```

```
        System.out.println( "C" );
    else if ( calificacionEstudiante >= 60 )
        System.out.println( "D" );
    else
        System.out.println( "F" );
```

Las dos formas son idénticas, excepto por el espaciado y la sangría, que el compilador ignora. La segunda forma es más popular ya que evita usar mucha sangría hacia la derecha en el código. Dicha sangría a menudo deja poco espacio en una línea de código, forzando a que las líneas se dividan y empeorando la legibilidad del programa.

Problema del `else` *suelto*

El compilador de Java siempre asocia un `else` con el `if` que le precede inmediatamente, a menos que se le indique otra cosa mediante la colocación de llaves (`{` y `}`). Este comportamiento puede ocasionar lo que se conoce como el **problema del `else` suelto**. Por ejemplo,

```
if ( x > 5 )
    if ( y > 5 )
        System.out.println( "x e y son > 5" );
else
    System.out.println( "x es <= 5" );
```

parece indicar que si `x` es mayor que `5`, la instrucción `if` anidada determina si `y` es también mayor que `5`. De ser así, se produce como resultado la cadena `"x e y son > 5"`. De lo contrario, parece ser que si `x` no es mayor que `5`, la instrucción `else` que es parte del `if...else` produce como resultado la cadena `"x es <= 5"`.

¡Cuidado! Esta instrucción `if...else` anidada no se ejecuta como parece ser. El compilador en realidad interpreta la instrucción así:

```
if ( x > 5 )
    if ( y > 5 )
        System.out.println( "x e y son > 5" );
else
    System.out.println( "x es <= 5" );
```

en donde el cuerpo del primer `if` es un `if...else` anidado. La instrucción `if` más externa evalúa si `x` es mayor que `5`. De ser así, la ejecución continúa evaluando si `y` es también mayor que `5`. Si la segunda condición es verdadera, se muestra la cadena apropiada (`"x e y son > 5"`). No obstante, si la segunda condición es falsa se muestra la cadena `"x es <= 5"`, aun cuando sabemos que `x` es mayor que `5`. Además, si la condición de la instrucción `if` exterior es falsa, se omite la instrucción `if...else` interior y no se muestra nada en pantalla.

Para forzar a que la instrucción `if...else` anidada se ejecute como se tenía pensado originalmente, debe escribirse de la siguiente manera:

```
if ( x > 5 )
{
    if ( y > 5 )
        System.out.println( "x e y son > 5" );
}
else
        System.out.println( "x es <= 5" );
```

Las llaves (`{}`) indican al compilador que la segunda instrucción `if` se encuentra en el cuerpo del primer `if`, y que el `else` está asociado con el *primer* `if`. Los ejercicios 4.27 y 4.28 analizan con más detalle el problema del `else` suelto.

Bloques

La instrucción `if` normalmente espera sólo una instrucción en su cuerpo. Para incluir varias instrucciones en el cuerpo de un `if` (o en el cuerpo del `else` en una instrucción `if...else`), encierre las instrucciones entre llaves (`{` y `}`). A un conjunto de instrucciones contenidas dentro de un par de llaves se le llama **bloque**. Un bloque puede colocarse en cualquier parte de un programa en donde pueda colocarse una sola instrucción.

El siguiente ejemplo incluye un bloque en la parte `else` de una instrucción `if...else`:

```
if ( calificacion >= 60 )
    System.out.println( "Aprobado" );
else
{
    System.out.println( "Reprobado." );
    System.out.println( "Debe tomar este curso otra vez." );
}
```

En este caso, si `calificacion` es menor que 60, el programa ejecuta ambas instrucciones en el cuerpo del `else` e imprime

```
Reprobado.
Debe tomar este curso otra vez.
```

Observe las llaves que rodean a las dos instrucciones en la cláusula `else`. Estas llaves son importantes. Sin ellas, la instrucción

```
System.out.println ( "Debe tomar este curso otra vez." );
```

estaría fuera del cuerpo de la parte `else` de la instrucción `if...else` y se ejecutaría sin importar que la calificación fuera menor a 60.

Los errores de sintaxis (como cuando se omite una llave en un bloque del programa) los atrapa el compilador. Un **error lógico** (como cuando se omiten ambas llaves en un bloque del programa) tiene su efecto en tiempo de ejecución. Un **error lógico fatal** hace que un programa falle y termine antes de tiempo. Un **error lógico no fatal** permite que un programa siga ejecutándose, pero éste produce resultados incorrectos.



Error común de programación 4.1

Olvidar una o las dos llaves que delimitan un bloque puede provocar errores de sintaxis o errores lógicos en un programa.



Buena práctica de programación 4.4

Colocar siempre las llaves en una instrucción `if...else` (o cualquier estructura de control) ayuda a evitar que se omitan de manera accidental, en especial, cuando posteriormente se agregan instrucciones a una cláusula `if` o `else`. Para evitar que esto suceda, algunos programadores prefieren escribir la llave inicial y la final de los bloques antes de escribir las instrucciones individuales dentro de ellas.

Así como un bloque puede colocarse en cualquier parte en donde pueda colocarse una sola instrucción individual, también es posible no tener instrucción alguna. En la sección 2.8 vimos que la instrucción vacía se representa colocando un punto y coma (`;`) en donde normalmente iría una instrucción.



Error común de programación 4.2

Colocar un punto y coma después de la condición en una instrucción `if...else` produce un error lógico en las instrucciones `if` de selección simple, y un error de sintaxis en las instrucciones `if...else` de selección doble (cuando la parte del `if` contiene una instrucción en el cuerpo).

4.7 Instrucción de repetición `while`

Una **instrucción de repetición** (también llamada **instrucción de ciclo**, o un **ciclo**) permite al programador especificar que un programa debe repetir una acción mientras cierta condición sea verdadera. La instrucción en seudocódigo

Mientras existan más artículos en mi lista de compras
Comprar el siguiente artículo y quitarlo de mi lista

describe la repetición que ocurre durante una salida de compras. La condición "existan más artículos en mi lista de compras" puede ser verdadera o falsa. Si es verdadera, entonces se realiza la acción "Comprar el siguiente artículo y quitarlo de mi lista". Esta acción se realizará en forma repetida mientras la condición sea verdadera. La instrucción

(o instrucciones) contenida en la instrucción de repetición *while* constituye el cuerpo de esta estructura, el cual puede ser una sola instrucción o un bloque. En algún momento, la condición será falsa (cuando el último artículo de la lista de compras sea adquirido y eliminado de la lista). En este punto la repetición terminará y se ejecutará la primera instrucción que esté después de la instrucción de repetición.

Como ejemplo de la instrucción de repetición `while` en Java, considere un segmento de programa diseñado para encontrar la primera potencia de 3 que sea mayor a 100. Suponga que la variable `producto` de tipo `int` se inicializa en 3. Cuando la siguiente instrucción `while` termine de ejecutarse, `producto` contendrá el resultado:

```
int producto = 3;

while ( producto <= 100 )
   producto = 3 * producto;
```

Cuando esta instrucción `while` comienza a ejecutarse, el valor de la variable `producto` es 3. Cada iteración de la instrucción `while` multiplica a `producto` por 3, por lo que `producto` toma los valores de 9, 27, 81 y 243, sucesivamente. Cuando la variable `producto` se vuelve 243, la condición de la instrucción `while` (`producto <= 1000`) se torna falsa. Esto termina la repetición, por lo que el valor final de `producto` es 243. En este punto, la ejecución del programa continúa con la siguiente instrucción después de la instrucción `while`.



Error común de programación 4.3

Si no se proporciona, en el cuerpo de una instrucción `while`*, una acción que ocasione que en algún momento la condición de un* `while` *se torne falsa, por lo general, se producirá un error lógico conocido como* ***ciclo infinito****, en el que el ciclo nunca terminará.*

El diagrama de actividad de UML de la figura 4.4 muestra el flujo de control que corresponde a la instrucción `while` anterior. Una vez más (aparte del estado inicial, las flechas de transición, un estado final y tres notas), los símbolos en el diagrama representan un estado de acción y una decisión. Este diagrama también introduce el **símbolo de fusión**. UML representa tanto al símbolo de fusión como al símbolo de decisión como rombos. El símbolo de fusión une dos flujos de actividad en uno solo. En este diagrama, el símbolo de fusión une las transiciones del estado inicial y del estado de acción, de manera que ambas fluyan en la decisión que determina si el ciclo debe empezar a ejecutarse (o seguir ejecutándose). Los símbolos de decisión y de fusión pueden diferenciarse por el número de flechas de transición "entrantes" y "salientes". Un símbolo de decisión tiene una flecha de transición que apunta hacia el rombo y dos o más flechas de transición que apuntan hacia fuera del rombo, para indicar las posibles transiciones desde ese punto. Además, cada flecha de transición que apunta hacia fuera de un símbolo de decisión tiene una condición de guardia junto a ella. Un símbolo de fusión tiene dos o más flechas de transición que apuntan hacia el rombo, y sólo una flecha de transición que apunta hacia fuera del rombo, para indicar múltiples flujos de actividad que se fusionan para continuar la actividad. Ninguna de las flechas de transición asociadas con un símbolo de fusión tiene una condición de guardia.

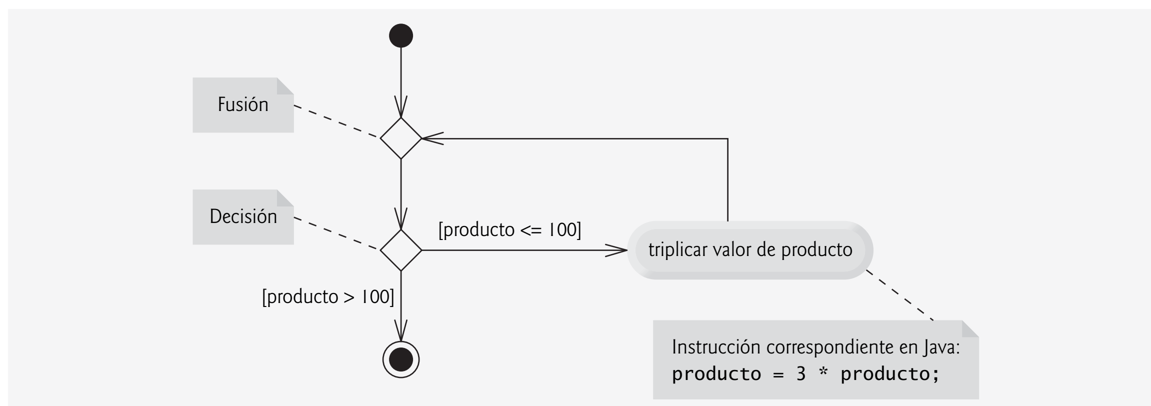


Figura 4.4 | Diagrama de actividad de UML de la instrucción de repetición `while`.

La figura 4.4 muestra claramente la repetición de la instrucción `while` que vimos antes en esta sección. La flecha de transición que emerge del estado de acción apunta de regreso a la fusión, desde la cual el flujo del programa regresa a la decisión que se evalúa al principio de cada iteración del ciclo. Éste ciclo sigue ejecutándose hasta que la condición de guardia `producto > 100` se vuelva verdadera. Entonces, la instrucción `while` termina (llega a su estado final) y el control pasa a la siguiente instrucción en la secuencia del programa.

4.8 Cómo formular algoritmos: repetición controlada por un contador

Para ilustrar la forma en que se desarrollan los algoritmos, modificamos la clase `LibroCalificaciones` del capítulo 3, para resolver dos variantes de un problema que promedia las calificaciones de unos estudiantes. Analicemos el siguiente enunciado del problema:

> *A una clase de diez estudiantes se les aplicó un examen. Las calificaciones (enteros en el rango de 0 a 100) de este examen están disponibles para su análisis. Determine el promedio de la clase para este examen.*

El promedio de la clase es igual a la suma de las calificaciones, dividida entre el número de estudiantes. El algoritmo para resolver este problema en una computadora debe recibir como entrada cada una de las calificaciones, llevar el registro del total de las calificaciones introducidas, realizar el cálculo para promediar e imprimir el resultado.

Algoritmo de seudocódigo con repetición controlada por un contador

Emplearemos seudocódigo para enlistar las acciones a ejecutar y especificar el orden en que deben ejecutarse. Usaremos una **repetición controlada por contador** para introducir las calificaciones, una por una. Esta técnica utiliza una variable llamada **contador** (o **variable de control**) para controlar el número de veces que debe ejecutarse un conjunto de instrucciones. A la repetición controlada por contador se le llama comúnmente **repetición definida**, ya que el número de repeticiones se conoce antes de que el ciclo comience a ejecutarse. En este ejemplo, la repetición termina cuando el contador excede a 10. Esta sección presenta un algoritmo de seudocódigo (figura 4.5) completamente desarrollado, y una versión de la clase `LibroCalificaciones` (figura 4.6) que implementa el algoritmo en un método de Java. Después presentamos una aplicación (figura 4.7) que demuestra el algoritmo en acción. En la sección 4.9 demostraremos cómo utilizar el seudocódigo para desarrollar dicho algoritmo desde cero.



Observación de ingeniería de software 4.1

La experiencia ha demostrado que la parte más difícil para la resolución de un problema en una computadora es desarrollar el algoritmo para la solución. Por lo general, una vez que se ha especificado el algoritmo correcto, el proceso de producir un programa funcional en Java a partir de dicho algoritmo es relativamente sencillo.

Observe las referencias en el algoritmo de la figura 4.5 para un total y un contador. Un **total** es una variable que se utiliza para acumular la suma de varios valores. Un contador es una variable que se utiliza para contar; en este caso, el contador de calificaciones indica cuál de las 10 calificaciones está a punto de escribir el usuario. Por lo general, las variables que se utilizan para guardar totales deben inicializarse en cero antes de utilizarse en un programa.

```
Asignar a total el valor de cero
Asignar al contador de calificaciones el valor de uno

Mientras que el contador de calificaciones sea menor o igual a diez
    Pedir al usuario que introduzca la siguiente calificación
    Obtener como entrada la siguiente calificación
    Sumar la calificación al total
    Sumar uno al contador de calificaciones

Asignar al promedio de la clase el total dividido entre diez
Imprimir el promedio de la clase
```

Figura 4.5 | Algoritmo en seudocódigo que utiliza la repetición controlada por contador para resolver el problema del promedio de una clase.

Implementación de la repetición controlada por contador en la clase `LibroCalificaciones`

La clase `LibroCalificaciones` (figura 4.6) contiene un constructor (líneas 11-14) que asigna un valor a la variable de instancia `nombreDelCurso` (declarada en la línea 8) de la clase. Las líneas 17 a la 20, 23 a la 26 y 29 a la 34 declaran los métodos `establecerNombreDelCurso`, `obtenerNombreDelCurso` y `mostrarMensaje`, respectivamente. Las líneas 37 a la 66 declaran el método `determinarPromedioClase`, el cual implementa el algoritmo para sacar el promedio de la clase, descrito por el seudocódigo de la figura 4.5.

La línea 40 declara e inicializa la variable `entrada` de tipo `Scanner`, que se utiliza para leer los valores introducidos por el usuario. Las líneas 42 a 45 declaran las variables locales `total`, `contadorCalif`, `calificacion` y `promedio` de tipo `int`. La variable `calificacion` almacena la entrada del usuario.

Observe que las declaraciones (en las líneas 42 a la 45) aparecen en el cuerpo del método `determinarPromedioClase`. Recuerde que las variables declaradas en el cuerpo de un método son variables locales, y sólo pueden utilizarse desde la línea de su declaración en el método, hasta la llave derecha de cierre (`}`) de la declaración del método. La declaración de una variable local debe aparecer antes de que la variable se utilice en ese método. Una variable local no puede utilizarse fuera del método en el que se declara.

En las versiones de la clase `LibroCalificaciones` en este capítulo, simplemente leemos y procesamos un conjunto de calificaciones. El cálculo del promedio se realiza en el método `determinarPromedioClase`, usando variables locales; no preservamos información acerca de las calificaciones de los estudiantes en variables de instancia de la clase. En versiones posteriores de la clase (en el capítulo 7, Arreglos), mantenemos las calificaciones en memoria utilizando una variable de instancia que hace referencia a una estructura de datos conocida como arreglo. Esto permite que un objeto `LibroCalificaciones` realice varios cálculos sobre el mismo conjunto de calificaciones, sin requerir que el usuario escriba las calificaciones varias veces.



Buena práctica de programación 4.5

Separe las declaraciones de las otras instrucciones en los métodos con una línea en blanco, para mejorar la legibilidad.

```
1  // Fig. 4.6: LibroCalificaciones.java
2  // La clase LibroCalificaciones que resuelve el problema del promedio de
3  // la clase, usando la repetición controlada por un contador.
4  import java.util.Scanner; // el programa utiliza la clase Scanner
5
6  public class LibroCalificaciones
7  {
8     private String nombreDelCurso; // el nombre del curso que representa este
                                     LibroCalificaciones
9
10    // el constructor inicializa a nombreDelCurso
11    public LibroCalificaciones( String nombre )
12    {
13       nombreDelCurso = nombre; // inicializa a nombreDelCurso
14    } // fin del constructor
15
16    // método para establecer el nombre del curso
17    public void establecerNombreDelCurso( String nombre )
18    {
19       nombreDelCurso = nombre; // almacena el nombre del curso
20    } // fin del método establecerNombreDelCurso
21
22    // método para obtener el nombre del curso
23    public String obtenerNombreDelCurso()
24    {
25       return nombreDelCurso;
26    } // fin del método obtenerNombreDelCurso
27
28    // muestra un mensaje de bienvenida al usuario de LibroCalificaciones
```

Figura 4.6 | Repetición controlada por contador: Problema del promedio de una clase. (Parte 1 de 2).

```
29      public void mostrarMensaje()
30      {
31         // obtenerNombreDelCurso obtiene el nombre del curso
32         System.out.printf( "Bienvenido al libro de calificaciones para\n%s!\n\n",
33            obtenerNombreDelCurso() );
34      } // fin del método mostrarMensaje
35
36      // determina el promedio de la clase, con base en las 10 calificaciones introducidas
           por el usuario
37      public void determinarPromedioClase()
38      {
39         // crea objeto Scanner para obtener la entrada de la ventana de comandos
40         Scanner entrada = new Scanner( System.in );
41
42         int total; // suma de las calificaciones escritas por el usuario
43         int contadorCalif; // número de la siguiente calificación a introducir
44         int calificacion; // valor de la calificación escrita por el usuario
45         int promedio; // el promedio de las calificaciones
46
47         // fase de inicialización
48         total = 0; // inicializa el total
49         contadorCalif = 1; // inicializa el contador del ciclo
50
51         // fase de procesamiento
52         while ( contadorCalif <= 10 ) // itera 10 veces
53         {
54            System.out.print( "Escriba la calificación: " ); // indicador
55            calificacion = entrada.nextInt(); // lee calificación del usuario
56            total = total + calificacion; // suma calificación a total
57            contadorCalif = contadorCalif + 1; // incrementa contador en 1
58         } // fin de while
59
60         // fase de terminación
61         promedio = total / 10; // la división entera produce un resultado entero
62
63         // muestra el total y el promedio de las calificaciones
64         System.out.printf( "\nEl total de las 10 calificaciones es %d\n", total );
65         System.out.printf( "El promedio de la clase es %d\n", promedio );
66      } // fin del método determinarPromedioClase
67
68   } // fin de la clase LibroCalificaciones
```

Figura 4.6 | Repetición controlada por contador: Problema del promedio de una clase. (Parte 2 de 2).

Las asignaciones (en las líneas 48 y 49) inicializan `total` a `0` y `contadorCalif` a `1`. Observe que estas inicializaciones ocurren antes que se utilicen las variables en los cálculos. Las variables `calificacion` y `promedio` (para la entrada del usuario y el promedio calculado, respectivamente) no necesitan inicializarse aquí; sus valores se asignarán a medida que se introduzcan o calculen más adelante en el método.



Error común de programación 4.4

Leer el valor de una variable local antes de inicializarla produce un error de compilación. Todas las variables locales deben inicializarse antes de leer sus valores en las expresiones.



Tip para prevenir errores 4.1

Inicialice cada contador y total, ya sea en su declaración o en una instrucción de asignación. Por lo general, los totales se inicializan a 0. Los contadores comúnmente se inicializan a 0 o a 1, dependiendo de cómo se utilicen (más adelante veremos ejemplos de cuándo usar 0 y cuándo usar 1).

La línea 52 indica que la instrucción `while` debe continuar ejecutando el ciclo (lo que también se conoce como **iterar**), siempre y cuando el valor de `contadorCalif` sea menor o igual a 10. Mientras esta condición sea verdadera, la instrucción `while` ejecutará en forma repetida las instrucciones entre las llaves que delimitan su cuerpo (líneas 54 a la 57).

La línea 54 muestra el indicador `"Escriba la calificacion: "`. La línea 55 lee el dato escrito por el usuario y lo asigna a la variable `calificacion`. Después, la línea 56 suma la nueva calificación escrita por el usuario al total, y asigna el resultado a `total`, que sustituye su valor anterior.

La línea 57 suma `1` a `contadorCalif` para indicar que el programa ha procesado una calificación y está listo para recibir la siguiente calificación del usuario. Al incrementar a `contadorCalif` en cada iteración, en un momento dado su valor excederá a 10. En ese momento, el ciclo `while` termina debido a que su condición (línea 52) se vuelve falsa.

Cuando el ciclo termina, la línea 61 realiza el cálculo del promedio y asigna su resultado a la variable `promedio`. La línea 64 utiliza el método `printf` de `System.out` para mostrar el texto `"El total de las 10 calificaciones es "`, seguido del valor de la variable `total`. Después, la línea 65 utiliza a `printf` para mostrar el texto `"El promedio de la clase es "`, seguido del valor de la variable `promedio`. Después de llegar a la línea 66, el método `determinarPromedioClase` devuelve el control al método que hizo la llamada (es decir, a `main` en `PruebaLibroCalificaciones` de la figura 4.7).

La clase `PruebaLibroCalificaciones`

La clase `PruebaLibroCalificaciones` (figura 4.7) crea un objeto de la clase `LibroCalificaciones` (figura 4.6) y demuestra sus capacidades. Las líneas 10 y 11 de la figura 4.7 crean un nuevo objeto `LibroCalificaciones` y lo asignan a la variable `miLibroCalificaciones`. El objeto `String` en la línea 11 se pasa al constructor de `LibroCalificaciones` (líneas 11 a la 14 de la figura 4.6). La línea 13 llama al método `mostrarMensaje` de `miLibroCalificaciones` para mostrar un mensaje de bienvenida al usuario. Después, la línea 14 llama al método `determinarPromedioClase` de `miLibroCalificaciones` para permitir que el usuario introduzca 10 calificaciones, para las cuales el método posteriormente calcula e imprime el promedio; el método ejecuta el algoritmo que se muestra en la figura 4.5.

Observaciones acerca de la división de enteros y el truncamiento

El cálculo del promedio realizado por el método `determinarPromedioClase`, en respuesta a la llamada al método en la línea 14 de la figura 4.7, produce un resultado entero. La salida del programa indica que la suma de los valores de las calificaciones en la ejecución de ejemplo es 846, que al dividirse entre 10, debe producir el número de punto flotante 84.6. Sin embargo, el resultado del cálculo `total / 10` (línea 61 de la figura 4.6) es el entero

```
1  // Fig. 4.7: PruebaLibroCalificaciones.java
2  // Crea un objeto LibroCalificaciones e invoca a su método obtenerPromedioClase.
3
4  public class PruebaLibroCalificaciones
5  {
6     public static void main( String args[] )
7     {
8        // crea objeto miLibroCalificaciones de la clase LibroCalificaciones y
9        // pasa el nombre del curso al constructor
10       LibroCalificaciones miLibroCalificaciones = new LibroCalificaciones(
11          "CS101 Introducción a la programación en Java" );
12
13       miLibroCalificaciones.mostrarMensaje(); // muestra mensaje de bienvenida
14       miLibroCalificaciones.determinarPromedioClase(); // encuentra el promedio de 10 calificaciones
15    } // fin de main
16
17 } // fin de la clase PruebaLibroCalificaciones
```

Figura 4.7 | La clase `PruebaLibroCalificaciones` crea un objeto de la clase `LibroCalificaciones` (figura 4.6) e invoca a su método `determinarPromedioClase`. (Parte 1 de 2).

```
Bienvenido al libro de calificaciones para
CS101 Introduccion a la programacion en Java!

Escriba la calificacion: 67
Escriba la calificacion: 78
Escriba la calificacion: 89
Escriba la calificacion: 67
Escriba la calificacion: 87
Escriba la calificacion: 98
Escriba la calificacion: 93
Escriba la calificacion: 85
Escriba la calificacion: 82
Escriba la calificacion: 100

El total de las 10 calificaciones es 846
El promedio de la clase es 84
```

Figura 4.7 | La clase `PruebaLibroCalificaciones` crea un objeto de la clase `LibroCalificaciones` (figura 4.6) e invoca a su método `determinarPromedioClase`. (Parte 2 de 2).

84, ya que `total` y `10` son enteros. Al dividir dos enteros se produce una **división entera**: se pierde cualquier parte fraccionaria del cálculo (es decir, se **trunca**). En la siguiente sección veremos cómo obtener un resultado de punto flotante a partir del cálculo del promedio.



Error común de programación 4.5

Suponer que la división entera redondea (en vez de truncar) puede producir resultados erróneos. Por ejemplo, 7 ÷ 4, que produce 1.75 en la aritmética convencional, se trunca a 1 en la aritmética entera, en vez de redondearse a 2.

4.9 Cómo formular algoritmos: repetición controlada por un centinela

Generalicemos el problema, de la sección 4.8, para los promedios de una clase. Considere el siguiente problema:

Desarrollar un programa que calcule el promedio de una clase y procese las calificaciones para un número arbitrario de estudiantes cada vez que se ejecute.

En el ejemplo anterior del promedio de una clase, el enunciado del problema especificó el número de estudiantes (10). En este ejemplo no se indica cuántas calificaciones introducirá el usuario durante la ejecución del programa. El programa debe procesar un número arbitrario de calificaciones. ¿Cómo puede el programa determinar cuándo terminar de introducir calificaciones? ¿Cómo sabrá cuándo calcular e imprimir el promedio de la clase?

Una manera de resolver este problema es utilizar un valor especial denominado **valor centinela** (también llamado **valor de señal**, **valor de prueba** o **valor de bandera**) para indicar el "fin de la introducción de datos". El usuario escribe calificaciones hasta que se haya introducido el número correcto de ellas. Después, el usuario escribe el valor centinela para indicar que no se van a introducir más calificaciones. A la repetición controlada por centinela a menudo se le llama **repetición indefinida**, ya que el número de repeticiones no se conoce antes de que comience la ejecución del ciclo.

Evidentemente, debe elegirse un valor centinela de tal forma que no pueda confundirse con un valor de entrada permitido. Las calificaciones de un examen son enteros positivos, por lo que –1 es un valor centinela aceptable para este problema. Por lo tanto, una ejecución del programa para promediar una clase podría procesar una cadena de entradas como 95, 96, 75, 74, 89 y –1. El programa entonces calcularía e imprimiría el promedio de la clase para las calificaciones 95, 96, 75, 74 y 89; como –1 es el valor centinela, no debe entrar en el cálculo del promedio.



Error común de programación 4.6

Seleccionar un valor centinela que sea también un valor de datos permitido es un error lógico.

Desarrollo del algoritmo en seudocódigo con el método de refinamiento de arriba a abajo, paso a paso: el primer refinamiento (cima)

Desarrollamos el programa para promediar clases con una técnica llamada **refinamiento de arriba a abajo**, **paso a paso**, la cual es esencial para el desarrollo de programas bien estructurados. Comenzamos con una representación en seudocódigo de la **cima**, una sola instrucción que transmite la función del programa en general:

Determinar el promedio de la clase para el examen

La cima es, en efecto, la representación *completa* de un programa. Desafortunadamente, la cima pocas veces transmite los detalles suficientes como para escribir un programa en Java. Por lo tanto, ahora comenzaremos el proceso de refinamiento. Dividiremos la cima en una serie de tareas más pequeñas y las enlistaremos en el orden en el que se van a realizar. Esto arroja como resultado el siguiente **primer refinamiento**:

Inicializar variables
Introducir, sumar y contar las calificaciones del examen
Calcular e imprimir el promedio de la clase

Esta mejora utiliza sólo la estructura de secuencia; los pasos aquí mostrados deben ejecutarse en orden, uno después del otro.



Observación de ingeniería de software 4.2

Cada mejora, así como la cima en sí, es una especificación completa del algoritmo; sólo varía el nivel del detalle.



Observación de ingeniería de software 4.3

Muchos programas pueden dividirse lógicamente en tres fases: de inicialización, en donde se inicializan las variables; procesamiento, en donde se introducen los valores de los datos y se ajustan las variables del programa (como contadores y totales) según sea necesario; y una fase de terminación, que calcula y produce los resultados finales.

Cómo proceder al segundo refinamiento

La anterior Observación de ingeniería de software es a menudo todo lo que usted necesita para el primer refinamiento en el proceso de arriba a abajo. Para avanzar al siguiente nivel de refinamiento, es decir, **el segundo refinamiento**, nos comprometemos a usar variables específicas. En este ejemplo necesitamos el total actual de los números, una cuenta de cuántos números se han procesado, una variable para recibir el valor de cada calificación, a medida que el usuario las vaya introduciendo, y una variable para almacenar el promedio calculado. La instrucción en seudocódigo

Inicializar las variables

puede mejorarse como sigue:

Inicializar total en cero
Inicializar contador en cero

Sólo las variables *total* y *contador* necesitan inicializase antes de que puedan utilizarse. Las variables *promedio* y *calificacion* (para el promedio calculado y la entrada del usuario, respectivamente) no necesitan inicializarse, ya que sus valores se reemplazarán a medida que se calculen o introduzcan.

La instrucción en seudocódigo

Introducir, sumar y contar las calificaciones del examen

requiere una estructura de repetición (es decir, un ciclo) que introduzca cada calificación en forma sucesiva. No sabemos de antemano cuántas calificaciones van a procesarse, por lo que utilizaremos la repetición controlada por centinela. El usuario introduce las calificaciones una por una; después de introducir la última calificación, introduce el valor centinela. El programa evalúa el valor centinela después de la introducción de cada calificación, y termina el ciclo cuando el usuario introduce el valor centinela. Entonces, la segunda mejora de la instrucción anterior en seudocódigo sería

Pedir al usuario que introduzca la primera calificación
Recibir como entrada la primera calificación (puede ser el centinela)

Mientras el usuario no haya introducido aún el centinela
    Sumar esta calificación al total actual
    Sumar uno al contador de calificaciones
    Pedir al usuario que introduzca la siguiente calificación
    Recibir como entrada la siguiente calificación (puede ser el centinela)

En seudocódigo no utilizamos llaves alrededor de las instrucciones que forman el cuerpo de la estructura *Mientras*. Simplemente aplicamos sangría a las instrucciones bajo el *Mientras* para mostrar que pertenecen a esta instrucción. De nuevo, el seudocódigo es solamente una herramienta informal para desarrollar programas.

La instrucción en seudocódigo

Calcular e imprimir el promedio de la clase

puede mejorarse de la siguiente manera:

Si el contador no es igual a cero
    Asignar al promedio el total dividido entre el contador
    Imprimir el promedio
De lo contrario
    Imprimir "No se introdujeron calificaciones"

Aquí tenemos cuidado de evaluar la posibilidad de una división entre cero; por lo general, esto es un error lógico que, si no se detecta, haría que el programa fallara o produjera resultados inválidos. El segundo refinamiento completo del seudocódigo para el problema del promedio de una clase se muestra en la figura 4.8.



Tip para prevenir errores 4.2

Al realizar una división entre una expresión cuyo valor pudiera ser cero, debe evaluar explícitamente esta posibilidad y manejarla de manera apropiada en su programa (como imprimir un mensaje de error), en vez de permitir que ocurra el error.

En las figuras 4.5 y 4.8 incluimos algunas líneas en blanco y sangría en el seudocódigo para facilitar su lectura. Las líneas en blanco separan los algoritmos en seudocódigo en sus diversas fases y accionan las instrucciones de control; la sangría enfatiza los cuerpos de las estructuras de control.

```
Inicializar total en cero
Inicializar contador en cero

Pedir al usuario que introduzca la primera calificación
Recibir como entrada la primera calificación (puede ser el centinela)

Mientras el usuario no haya introducido aún el centinela
    Sumar esta calificación al total actual
    Sumar uno al contador de calificaciones
    Pedir al usuario que introduzca la siguiente calificación
    Recibir como entrada la siguiente calificación (puede ser el centinela)

Si el contador no es igual a cero
    Asignar al promedio el total dividido entre el contador
    Imprimir el promedio
De lo contrario
    Imprimir "No se introdujeron calificaciones"
```

Figura 4.8 | Algoritmo en seudocódigo del problema para promediar una clase, con una repetición controlada por centinela.

El algoritmo en seudocódigo en la figura 4.8 resuelve el problema más general para promediar una clase. Este algoritmo se desarrolló después de aplicar dos niveles de refinamiento. En ocasiones se requieren más niveles de refinamiento.



Observación de ingeniería de software 4.4

Termine el proceso de refinamiento de arriba a abajo, paso a paso, cuando haya especificado el algoritmo en seudocódigo con el detalle suficiente como para poder convertir el seudocódigo en Java. Por lo general, la implementación del programa en Java después de esto es mucho más sencilla.



Observación de ingeniería de software 4.5

Algunos programadores experimentados escriben programas sin utilizar herramientas de desarrollo de programas como el seudocódigo. Estos programadores sienten que su meta final es resolver el problema en una computadora y que el escribir seudocódigo simplemente retarda la producción de los resultados finales. Aunque este método pudiera funcionar para problemas sencillos y conocidos, tiende a ocasionar graves errores y retrasos en proyectos grandes y complejos.

Implementación de la repetición controlada por centinela en la clase `LibroCalificaciones`

La figura 4.9 muestra la clase de Java `LibroCalificaciones` que contiene el método `determinarPromedioClase`, el cual implementa el algoritmo, de la figura 4.8, en seudocódigo. Aunque cada calificación es un valor entero, existe la probabilidad de que el cálculo del promedio produzca un número con un punto decimal; en otras palabras, un número real (es decir, de punto flotante). El tipo `int` no puede representar un número de este tipo, por lo que esta clase utiliza el tipo `double` para ello.

En este ejemplo vemos que las estructuras de control pueden apilarse una encima de otra (en secuencia), al igual que un niño apila bloques de construcción. La instrucción `while` (líneas 57 a 65) va seguida por una instrucción `if...else` (líneas 69 a 80) en secuencia. La mayor parte del código en este programa es igual al código de la figura 4.6, por lo que nos concentraremos en los nuevos conceptos.

La línea 45 declara la variable `promedio` de tipo `double`, la cual nos permite guardar el promedio de la clase como un número de punto flotante. La línea 49 inicializa `contadorCalif` en `0`, ya que todavía no se han introducido calificaciones. Recuerde que este programa utiliza la repetición controlada por centinela para recibir las calificaciones que escribe el usuario. Para mantener un registro preciso del número de calificaciones introducidas, el programa incrementa `contadorCalif` sólo cuando el usuario introduce un valor permitido para la calificación.

```
1  // Fig. 4.9: LibroCalificaciones.java
2  // La clase LibroCalificaciones resuelve el problema del promedio de la clase
3  // usando la repetición controlada por un centinela.
4  import java.util.Scanner; // el programa usa la clase Scanner
5
6  public class LibroCalificaciones
7  {
8     private String nombreDelCurso; // el nombre del curso que representa este
                                        LibroCalificaciones
9
10    // el constructor inicializa a nombreDelCurso
11    public LibroCalificaciones( String nombre )
12    {
13       nombreDelCurso = nombre; // inicializa a nombreDelCurso
14    } // fin del constructor
15
16    // método para establecer el nombre del curso
17    public void establecerNombreDelCurso( String nombre )
18    {
```

Figura 4.9 | Repetición controlada por centinela: problema del promedio de una clase. (Parte 1 de 3).

```
      nombreDelCurso = nombre; // almacena el nombre del curso
   } // fin del método establecerNombreDelCurso

   // método para obtener el nombre del curso
   public String obtenerNombreDelCurso()
   {
      return nombreDelCurso;
   } // fin del método obtenerNombreDelCurso

   // muestra un mensaje de bienvenida al usuario de LibroCalificaciones
   public void mostrarMensaje()
   {
      // obtenerNombreDelCurso obtiene el nombre del curso
      System.out.printf( "Bienvenido al libro de calificaciones para\n%s!\n\n",
         obtenerNombreDelCurso() );
   } // fin del método mostrarMensaje

   // determina el promedio de un número arbitrario de calificaciones
   public void determinarPromedioClase()
   {
      // crea objeto Scanner para obtener la entrada de la ventana de comandos
      Scanner entrada = new Scanner( System.in );

      int total; // suma de las calificaciones
      int contadorCalif; // número de calificaciones introducidas
      int calificacion; // valor de calificación
      double promedio; // número con punto decimal para el promedio

      // fase de inicialización
      total = 0; // inicializa el total
      contadorCalif = 0; // inicializa el contador del ciclo

      // fase de procesamiento
      // pide entrada y lee calificación del usuario
      System.out.print( "Escriba calificacion o -1 para terminar: " );
      calificacion = entrada.nextInt();

      // itera hasta leer el valor centinela del usuario
      while ( calificacion != -1 )
      {
         total = total + calificacion; // suma calificacion al total
         contadorCalif = contadorCalif + 1; // incrementa el contador

         // pide entrada y lee siguiente calificación del usuario
         System.out.print( "Escriba calificacion o -1 para terminar: " );
         calificacion = entrada.nextInt();
      } // fin de while

      // fase de terminación
      // si el usuario introdujo por lo menos una calificación...
      if ( contadorCalif != 0 )
      {
         // calcula el promedio de todas las calificaciones introducidas
         promedio = (double) total / contadorCalif;

         // muestra el total y el promedio (con dos dígitos de precisión)
         System.out.printf( "\nEl total de las %d calificaciones introducidas es %d\n",
            contadorCalif, total );
         System.out.printf( "El promedio de la clase es %.2f\n", promedio );
```

Figura 4.9 | Repetición controlada por centinela: problema del promedio de una clase. (Parte 2 de 3).

```
78          } // fin de if
79          else // no se introdujeron calificaciones, por lo que se imprime el mensaje
                    apropiado
80             System.out.println( "No se introdujeron calificaciones" );
81       } // fin del método determinarPromedioClase
82
83    } // fin de la clase LibroCalificaciones
```

Figura 4.9 | Repetición controlada por centinela: problema del promedio de una clase. (Parte 3 de 3).

Comparación entre la lógica del programa para la repetición controlada por centinela, y la repetición controlada por contador

Compare la lógica de esta aplicación para la repetición controlada por centinela con la repetición controlada por contador en la figura 4.6. En la repetición controlada por contador, cada iteración de la instrucción `while` (líneas 52 a 58 de la figura 4.6) lee un valor del usuario, para el número especificado de iteraciones. En la repetición controlada por centinela, el programa lee el primer valor (líneas 53 y 54 de la figura 4.9) antes de llegar al `while`. Este valor determina si el flujo de control del programa debe entrar al cuerpo del `while`. Si la condición del `while` es falsa, el usuario introdujo el valor centinela, por lo que el cuerpo del `while` no se ejecuta (es decir, no se introdujeron calificaciones). Si, por otro lado, la condición es verdadera, el cuerpo comienza a ejecutarse y el ciclo suma el valor de `calificacion` al `total` (línea 59). Después, las líneas 63 y 64 en el cuerpo del ciclo reciben el siguiente valor escrito por el usuario. A continuación, el control del programa se acerca a la llave derecha de terminación (`}`) del cuerpo del ciclo en la línea 65, por lo que la ejecución continúa con la evaluación de la condición del `while` (línea 57). La condición utiliza el valor más reciente de `calificacion` que acaba de introducir el usuario, para determinar si el cuerpo de la instrucción `while` debe ejecutarse otra vez. Observe que el valor de la variable `calificacion` siempre lo introduce el usuario inmediatamente antes de que el programa evalúe la condición del `while`. Esto permite al programa determinar si el valor que acaba de introducir el usuario es el valor centinela, *antes* de que el programa procese ese valor (es decir, que lo sume al `total`). Si el valor introducido es el valor centinela, el ciclo termina y el programa no suma –1 al `total`.



Buena práctica de programación 4.6

En un ciclo controlado por centinela, los indicadores que solicitan la introducción de datos deben recordar explícitamente al usuario el valor que representa al centinela.

Una vez que termina el ciclo se ejecuta la instrucción `if...else` en las líneas 69 a 80. La condición en la línea 69 determina si se introdujeron calificaciones o no. Si no se introdujo ninguna, se ejecuta la parte del `else` (líneas 79 y 80) de la instrucción `if...else` y muestra el mensaje `"No se introdujeron calificaciones"`, y el método devuelve el control al método que lo llamó.

Observe el bloque de la instrucción `while` en la figura 4.9 (líneas 58 a 65). Sin las llaves, el ciclo consideraría que su cuerpo sólo consiste en la primera instrucción, que suma la `calificacion` al `total`. Las últimas tres instrucciones en el bloque quedarían fuera del cuerpo del ciclo, ocasionando que la computadora interpretara el código incorrectamente, como se muestra a continuación:

```
while ( calificacion != -1 )
   total = total + calificacion; // suma calificación al total
contadorCalif = contadorCalif + 1; // incrementa el contador

// obtiene como entrada la siguiente calificación del usuario
System.out.print( "Escriba calificacion o -1 para terminar: " );
calificacion = entrada.nextInt();
```

El código anterior ocasionaría un ciclo infinito en el programa, si el usuario no introduce el centinela `-1` como valor de entrada en la línea 54 (antes de la instrucción `while`).



Error común de programación 4.7

Omitir las llaves que delimitan a un bloque puede provocar errores lógicos, como ciclos infinitos. Para prevenir este problema, algunos programadores encierran el cuerpo de todas las instrucciones de control con llaves, aun si el cuerpo sólo contiene una instrucción.

Conversión explícita e implícita entre los tipos primitivos

Si se introdujo por lo menos una calificación, la línea 72 de la figura 4.9 calcula el promedio de las calificaciones. En la figura 4.6 vimos que la división entera produce un resultado entero. Aun y cuando la variable `promedio` se declara como `double` (línea 45), el cálculo

```
promedio = total / contadorCalif;
```

descarta la parte fraccionaria del cociente antes de asignar el resultado de la división a `promedio`. Esto ocurre debido a que `total` y `contadorCalif` son enteros, y la división entera produce un resultado entero. Para realizar un cálculo de punto flotante con valores enteros, debemos tratar temporalmente a estos valores como números de punto flotante, para usarlos en el cálculo. Java cuenta con el **operador unario de conversión de tipo** para llevar a cabo esta tarea. La línea 72 utiliza el operador de conversión de tipo **(double)** (un operador unario) para crear una copia de punto flotante *temporal* de su operando `total` (que aparece a la derecha del operador). Utilizar un operador de conversión de tipo de esta forma es un proceso que se denomina **conversión explícita**. El valor almacenado en `total` sigue siendo un entero.

El cálculo ahora consiste de un valor de punto flotante (la versión temporal `double` de `total`) dividido entre el entero `contadorCalif`. Java sabe cómo evaluar sólo expresiones aritméticas en las que los tipos de los operandos sean idénticos. Para asegurar que los operandos sean del mismo tipo, Java realiza una operación llamada **promoción** (o **conversión implícita**) en los operandos seleccionados. Por ejemplo, en una expresión que contenga valores de los tipos `int` y `double`, los valores `int` son **promovidos** a valores `double` para utilizarlos en la expresión. En este ejemplo, Java promueve el valor de `contadorCalif` al tipo `double`, después el programa realiza la división de punto flotante y asigna el resultado del cálculo a `promedio`. Mientras que se aplique el operador de conversión de tipo `(double)` a cualquier variable en el cálculo, éste producirá un resultado `double`. Más adelante en el capítulo, hablaremos sobre todos los tipos primitivos. En la sección 6.7 aprenderá más acerca de las reglas de promoción.



Error común de programación 4.8

El operador de conversión de tipo puede utilizarse para convertir entre los tipos numéricos primitivos, como `int` *y* `double`*, y para convertir entre los tipos de referencia relacionados (como lo describiremos en el capítulo 10, Programación orientada a objetos: polimorfismo). La conversión al tipo incorrecto puede ocasionar errores de compilación o errores en tiempo de ejecución.*

Los operadores de conversión de tipo están disponibles para cualquier tipo. El operador de conversión se forma colocando paréntesis alrededor del nombre de un tipo. Este operador es un **operador unario** (es decir, un operador que utiliza sólo un operando). En el capítulo 2 estudiamos los operadores aritméticos binarios. Java también soporta las versiones unarias de los operadores de suma (+) y resta (-), por lo que el programador puede escribir expresiones como `-7` o `+5`. Los operadores de conversión de tipo se asocian de derecha a izquierda y tienen la misma precedencia que los demás operadores unarios, como `+` y `-`. Esta precedencia es un nivel mayor que la de los **operadores de multiplicación** `*`, `/` y `%`. (Consulte la tabla de precedencia de operadores en el apéndice A). En nuestras tablas de precedencia, indicamos el operador de conversión de tipos con la notación (*tipo*) para indicar que puede usarse cualquier nombre de tipo para formar un operador de conversión de tipo.

La línea 77 imprime el promedio de la clase, usando el método `printf` de `System.out`. En este ejemplo mostramos el promedio de la clase redondeado a la centésima más cercana. El especificador de formato `%.2f` en la cadena de control de formato de `printf` (línea 77) indica que el valor de la variable `promedio` debe mostrarse con dos dígitos de precisión a la derecha del punto decimal; esto se indica mediante el `.2` en el especificador de formato. Las tres calificaciones introducidas durante la ejecución de ejemplo de la clase `PruebaLibroCalificaciones` (figura 4.10) dan un total de 257, que produce el promedio de 85.666666…. El método `printf` utiliza la precisión en el especificador de formato para redondear el valor al número especificado de dígitos. En este programa, el promedio se redondea a la posición de las centésimas y se muestra como 85.67.

```
// Fig. 4.10: PruebaLibroCalificaciones.java
// Crea un objeto LibroCalificaciones e invoca a su método determinarPromedioClase.

public class PruebaLibroCalificaciones
{
   public static void main( String args[] )
   {
      // crea objeto miLibroCalificaciones de LibroCalificaciones y
      // pasa el nombre del curso al constructor
      LibroCalificaciones miLibroCalificaciones = new LibroCalificaciones(
         "CS101 Introduccion a la programacion en Java" );

      miLibroCalificaciones.mostrarMensaje(); // muestra mensaje de bienvenida
      miLibroCalificaciones.determinarPromedioClase(); // encuentra el promedio de las
                                                          calificaciones
   } // fin de main

} // fin de la clase PruebaLibroCalificaciones
```

```
Bienvenido al libro de calificaciones para
CS101 Introduccion a la programacion en Java!

Escriba calificacion o -1 para terminar: 97
Escriba calificacion o -1 para terminar: 88
Escriba calificacion o -1 para terminar: 72
Escriba calificacion o -1 para terminar: -1

El total de las 3 calificaciones introducidas es 257
El promedio de la clase es 85.67
```

Figura 4.10 | La clase `PruebaLibroCalificaciones` crea un objeto de la clase `LibroCalificaciones` (figura 4.9) e invoca al método `determinarPromedioClase`.

4.10 Cómo formular algoritmos: instrucciones de control anidadas

En el siguiente ejemplo formularemos una vez más un algoritmo utilizando seudocódigo y el refinamiento de arriba a abajo, paso a paso, y después escribiremos el correspondiente programa en Java. Hemos visto que las instrucciones de control pueden apilarse una encima de otra (en secuencia). En este ejemplo práctico examinaremos la otra forma en la que pueden conectarse las instrucciones de control, a saber, mediante el **anidamiento** de una instrucción de control dentro de otra.

Considere el siguiente enunciado de un problema:

Una universidad ofrece un curso que prepara a los estudiantes para el examen estatal de certificación del estado como corredores de bienes raíces. El año pasado, diez de los estudiantes que completaron este curso tomaron el examen. La universidad desea saber qué tan bien se desempeñaron sus estudiantes en el examen. A usted se le ha pedido que escriba un programa para sintetizar los resultados. Se le dio una lista de estos 10 estudiantes. Junto a cada nombre hay un 1 escrito, si el estudiante aprobó el examen, o un 2 si lo reprobó.

Su programa debe analizar los resultados del examen de la siguiente manera:

1. *Introducir cada resultado de la prueba (es decir, un 1 o un 2). Mostrar el mensaje "Escriba el resultado" en la pantalla, cada vez que el programa solicite otro resultado de la prueba.*
2. *Contar el número de resultados de la prueba, de cada tipo.*
3. *Mostrar un resumen de los resultados de la prueba, indicando el número de estudiantes que aprobaron y el número de estudiantes que reprobaron.*
4. *Si más de ocho estudiantes aprobaron el examen, imprimir el mensaje "Aumentar la colegiatura".*

Después de leer el enunciado del programa cuidadosamente, hacemos las siguientes observaciones:

1. El programa debe procesar los resultados de la prueba para 10 estudiantes. Puede usarse un ciclo controlado por contador, ya que el número de resultados de la prueba se conoce de antemano.
2. Cada resultado de la prueba tiene un valor numérico, ya sea 1 o 2. Cada vez que el programa lee un resultado de la prueba, debe determinar si el número es 1 o 2. Nosotros evaluamos un 1 en nuestro algoritmo. Si el número no es 1, suponemos que es un 2. (El ejercicio 4.24 considera las consecuencias de esta suposición).
3. Dos contadores se utilizan para llevar el registro de los resultados del examen: uno para contar el número de estudiantes que aprobaron el examen y uno para contar el número de estudiantes que reprobaron el examen.
4. Una vez que el programa ha procesado todos los resultados, debe decidir si más de ocho estudiantes aprobaron el examen.

Veamos ahora el refinamiento de arriba a abajo, paso a paso. Comencemos con la representación del seudocódigo de la cima:

Analizar los resultados del examen y decidir si debe aumentarse la colegiatura o no.

Una vez más, la cima es una representación *completa* del programa, pero es probable que se necesiten varios refinamientos antes de que el seudocódigo pueda evolucionar de manera natural en un programa en Java.

Nuestro primer refinamiento es

Inicializar variables
Introducir las 10 calificaciones del examen y contar los aprobados y reprobados
Imprimir un resumen de los resultados del examen y decidir si debe aumentarse la colegiatura

Aquí también, aun cuando tenemos una representación completa del programa, es necesario refinarla. Ahora nos comprometemos con variables específicas. Se necesitan contadores para registrar los aprobados y reprobados; utilizaremos un contador para controlar el proceso de los ciclos y necesitaremos una variable para guardar la entrada del usuario. La variable en la que se almacenará la entrada del usuario no se inicializa al principio del algoritmo, ya que su valor proviene del usuario durante cada iteración del ciclo.

La instrucción en seudocódigo

Inicializar variables

puede mejorarse de la siguiente manera:

Inicializar aprobados en cero
Inicializar reprobados en cero
Inicializar contador de estudiantes en cero

Observe que sólo se inicializan los contadores al principio del algoritmo.

La instrucción en seudocódigo

Introducir las 10 calificaciones del examen, y contar los aprobados y reprobados

requiere un ciclo en el que se introduzca sucesivamente el resultado de cada examen. Sabemos de antemano que hay precisamente 10 resultados del examen, por lo que es apropiado utilizar un ciclo controlado por contador. Dentro del ciclo (es decir, anidado dentro del ciclo), una estructura de selección doble determinará si cada resultado del examen es aprobado o reprobado, e incrementará el contador apropiado. Entonces, la mejora al seudocódigo anterior es

Mientras el contador de estudiantes sea menor o igual a 10
    Pedir al usuario que introduzca el siguiente resultado del examen
    Recibir como entrada el siguiente resultado del examen

    Si el estudiante aprobó
        Sumar uno a aprobados

De lo contrario
Sumar uno a reprobados
Sumar uno al contador de estudiantes

Nosotros utilizamos líneas en blanco para aislar la estructura de control *Si...De lo contrario*, lo cual mejora la legibilidad.

La instrucción en seudocódigo

Imprimir un resumen de los resultados de los exámenes y decidir si debe aumentarse la colegiatura

puede mejorarse de la siguiente manera:

Imprimir el número de aprobados
Imprimir el número de reprobados

Si más de ocho estudiantes aprobaron
Imprimir "Aumentar la colegiatura"

Segundo refinamiento completo en seudocódigo y conversión a la clase `Analisis`

El segundo refinamiento completo aparece en la figura 4.11. Observe que también se utilizan líneas en blanco para separar la estructura *Mientras* y mejorar la legibilidad del programa. Este seudocódigo está ahora lo suficientemente mejorado para su conversión a Java. La clase de Java que implementa el algoritmo en seudocódigo se muestra en la figura 4.12, y en la figura 4.13 aparecen dos ejecuciones de ejemplo.

Las líneas 13 a 16 de la figura 4.12 declaran las variables que utiliza el método `procesarResultadosExamen` de la clase `Analisis` para procesar los resultados del examen. Varias de estas declaraciones utilizan la habilidad de Java para incorporar la inicialización de variables en las declaraciones (a `aprobados` se le asigna `0`, a `reprobados` se le asigna `0` y a `contadorEstudiantes` se le asigna `1`). Los programas con ciclos pueden requerir de la inicialización al principio de cada repetición; por lo general, dicha reinicialización se realiza mediante instrucciones de asignación, en vez de hacerlo en las declaraciones.

La instrucción `while` (líneas 19 a 33) itera 10 veces. Durante cada iteración, el ciclo recibe y procesa un resultado del examen. Observe que la instrucción `if...else` (líneas 26 a 29) para procesar cada resultado se anida en la instrucción `while`. Si `resultado` es `1`, la instrucción `if...else` incrementa a `aprobados`; en caso

```
Inicializar aprobados en cero
Inicializar reprobados en cero
Inicializar contador de estudiantes en uno

Mientras el contador de estudiantes sea menor o igual a 10
    Pedir al usuario que introduzca el siguiente resultado del examen
    Recibir como entrada el siguiente resultado del examen

    Si el estudiante aprobó
        Sumar uno a aprobados
    De lo contrario
        Sumar uno a reprobados

    Sumar uno al contador de estudiantes

Imprimir el número de aprobados
Imprimir el número de reprobados

Si más de ocho estudiantes aprobaron
    Imprimir "Aumentar colegiatura"
```

Figura 4.11 | El seudocódigo para el problema de los resultados del examen.

```
1  // Fig. 4.12: Analisis.java
2  // Análisis de los resultados de un examen.
3  import java.util.Scanner; // esta clase utiliza la clase Scanner
4
5  public class Analisis
6  {
7     public void procesarResultadosExamen()
8     {
9        // crea objeto Scanner para obtener la entrada de la ventana de comandos
10       Scanner entrada = new Scanner( System.in );
11
12       // inicialización de las variables en declaraciones
13       int aprobados = 0; // número de aprobados
14       int reprobados = 0; // número de reprobados
15       int contadorEstudiantes = 1; // contador de estudiantes
16       int resultado; // un resultado del examen (obtiene el valor del usuario)
17
18       // procesa 10 estudiantes, usando ciclo controlado por contador
19       while ( contadorEstudiantes <= 10 )
20       {
21          // pide al usuario la entrada y obtiene el valor
22          System.out.print( "Escriba el resultado (1 = aprobado, 2 = reprobado): " );
23          resultado = entrada.nextInt();
24
25          // if...else anidado en while
26          if ( resultado == 1 )           // si resultado 1,
27             aprobados = aprobados + 1;   // incrementa aprobados;
28          else                            // de lo contrario, resultado no es 1, por lo que
29             reprobados = reprobados + 1; // incrementa reprobados
30
31          // incrementa contadorEstudiantes, para que el ciclo termine en un momento dado
32          contadorEstudiantes = contadorEstudiantes + 1;
33       } // fin de while
34
35       // fase de terminación; prepara y muestra los resultados
36       System.out.printf( "Aprobados: %d\nReprobados: %d\n", aprobados, reprobados );
37
38       // determina si más de 8 estudiantes aprobaron
39       if ( aprobados > 8 )
40          System.out.println( "Aumentar colegiatura" );
41    } // fin del método procesarResultadosExamen
42
43 } // fin de la clase Analisis
```

Figura 4.12 | Estructuras de control anidadas: problema de los resultados del examen.

contrario, asume que `resultado` es `2` e incrementa `reprobados`. La línea 32 incrementa `contadorEstudiantes` antes de que se evalúe otra vez la condición del ciclo, en la línea 19. Después de introducir 10 valores, el ciclo termina y la línea 36 muestra el número de `aprobados` y de `reprobados`. La instrucción `if` de las líneas 39 a 40 determina si más de ocho estudiantes aprobaron el examen y, de ser así, imprime el mensaje `"Aumentar colegiatura"`.



Tip para prevenir errores 4.3

Inicializar las variables locales cuando se declaran ayuda al programador a evitar cualquier error de compilación que pudiera surgir, debido a los intentos por utilizar datos sin inicializar. Aunque Java no requiere que se incorporen las inicializaciones de variables locales en las declaraciones, sí requiere que se inicialicen las variables locales antes de utilizar sus valores en una expresión.

La clase `PruebaAnalisis` para demostrar la clase `Analisis`

La clase `PruebaAnalisis` (figura 4.13) crea un objeto `Analisis` (línea 8) e invoca al método `procesarResultadosExamen` (línea 9) de ese objeto para procesar un conjunto de resultados de un examen, introducidos por el usuario. La figura 4.13 muestra la entrada y salida de dos ejecuciones de ejemplo del programa. Durante la primera ejecución de ejemplo, la condición en la línea 39 del método `procesarResultadosExamen` de la figura 4.12 es verdadera; más de ocho estudiantes aprobaron el examen, por lo que el programa imprime un mensaje indicando que se debe aumentar la colegiatura.

```
1   // Fig. 4.13: PruebaAnalisis.java
2   // Programa de prueba para la clase Analisis.
3
4   public class PruebaAnalisis
5   {
6      public static void main( String args[] )
7      {
8         Analisis aplicacion = new Analisis(); // crea objeto Analisis
9         aplicacion.procesarResultadosExamen(); // llama al método para procesar los
                                                    resultados
10     } // fin de main
11
12  } // fin de la clase PruebaAnalisis
```

```
Escriba el resultado (1 = aprobado, 2 = reprobado): 1
Escriba el resultado (1 = aprobado, 2 = reprobado): 2
Escriba el resultado (1 = aprobado, 2 = reprobado): 1
Escriba el resultado (1 = aprobado, 2 = reprobado): 1
Escriba el resultado (1 = aprobado, 2 = reprobado): 1
Escriba el resultado (1 = aprobado, 2 = reprobado): 1
Escriba el resultado (1 = aprobado, 2 = reprobado): 1
Escriba el resultado (1 = aprobado, 2 = reprobado): 1
Escriba el resultado (1 = aprobado, 2 = reprobado): 1
Escriba el resultado (1 = aprobado, 2 = reprobado): 1
Aprobados: 9
Reprobados: 1
Aumentar colegiatura

Escriba el resultado (1 = aprobado, 2 = reprobado): 1
Escriba el resultado (1 = aprobado, 2 = reprobado): 2
Escriba el resultado (1 = aprobado, 2 = reprobado): 1
Escriba el resultado (1 = aprobado, 2 = reprobado): 2
Escriba el resultado (1 = aprobado, 2 = reprobado): 1
Escriba el resultado (1 = aprobado, 2 = reprobado): 2
Escriba el resultado (1 = aprobado, 2 = reprobado): 2
Escriba el resultado (1 = aprobado, 2 = reprobado): 1
Escriba el resultado (1 = aprobado, 2 = reprobado): 1
Escriba el resultado (1 = aprobado, 2 = reprobado): 1
Aprobados: 6
Reprobados: 4
```

Figura 4.13 | Programa de prueba para la clase `Analisis` (figura 4.12).

4.11 Operadores de asignación compuestos

Java cuenta con varios **operadores de asignación compuestos** para abreviar las expresiones de asignación. Cualquier instrucción de la forma

variable = *variable* *operador* *expresión*;

en donde *operador* es uno de los operadores binarios +, -, *, / o % (o alguno de los otros que veremos más adelante en el libro), puede escribirse de la siguiente forma:

variable *operador*= *expresión*;

Por ejemplo, puede abreviar la instrucción

```
c = c + 3;
```

mediante el **operador de asignación compuesto de suma**, +=, de la siguiente manera:

```
c += 3;
```

El operador += suma el valor de la expresión que está a la derecha del operador, al valor de la variable que está a la izquierda del operador, y almacena el resultado en la variable que está a la izquierda del operador. Por lo tanto, la expresión de asignación c += 3 suma 3 a c. La figura 4.14 muestra los operadores de asignación aritméticos compuestos, algunas expresiones de ejemplo en las que se utilizan los operadores y las explicaciones de lo que estos operadores hacen.

Operador de asignación	Expresión de ejemplo	Explicación	Asigna
Suponer que: int c = 3, d = 5, e = 4, f = 6, g = 12;			
+=	c += 7	c = c + 7	10 a c
-=	d -= 4	d = d - 4	1 a d
*=	e *= 5	e = e * 5	20 a e
/=	f /= 3	f = f / 3	2 a f
%=	g %= 9	g = g % 9	3 a g

Figura 4.14 | Operadores de asignación aritméticos.

4.12 Operadores de incremento y decremento

Java proporciona dos operadores unarios para sumar 1, o restar 1, al valor de una variable numérica. Estos operadores son el **operador de incremento** unario, ++, y el **operador de decremento** unario, --, los cuales se sintetizan en la figura 4.15. Un programa puede incrementar en 1 el valor de una variable llamada c, utilizando el operador de incremento, ++, en lugar de usar la expresión c = c + 1 o c += 1. A un operador de incremento o decremento que se coloca antes de una variable se le llama **operador de preincremento** o **predecremento**, respectivamente. A un operador de incremento o decremento que se coloca después de una variable se le llama **operador de postincremento** o **postdecremento**, respectivamente.

Al proceso de utilizar el operador de preincremento (o postdecremento) para sumar (o restar) 1 a una variable, se le conoce como **preincrementar** (o **predecrementar**) la variable. Al preincrementar (o predecrementar) una variable, ésta se incrementa (o decrementa) en 1, y después el nuevo valor de la variable se utiliza en la expresión en la que aparece. Al proceso de utilizar el operador de preincremento (o postdecremento) para sumar (o restar) 1 a una variable, se le conoce como **postincrementar** (o **postdecrementar**) la variable. Al postincrementar (o postdecrementar) una variable, el valor actual de la variable se utiliza en la expresión en la que aparece y después el valor de la variable se incrementa (o decrementa) en 1.



Buena práctica de programación 4.7

A diferencia de los operadores binarios, los operadores unarios de incremento y decremento deben colocarse enseguida de sus operandos, sin espacios entre ellos.

Operador	Operador Llamado	Expresión de ejemplo	Explicación
++	Preincremento	++a	Incrementar a en 1, después utilizar el nuevo valor de a en la expresión en que esta variable reside.
++	Postincremento	a++	Usar el valor actual de a en la expresión en la que esta variable reside, después incrementar a en 1.
--	Predecremento	--b	Decrementar b en 1, después utilizar el nuevo valor de b en la expresión en que esta variable reside.
--	Postdecremento	b--	Usar el valor actual de b en la expresión en la que esta variable reside, después decrementar b en 1.

Figura 4.15 | Los operadores de incremento y decremento.

La figura 4.16 demuestra la diferencia entre la versión de preincremento y la versión de predecremento del operador de incremento ++. El operador de decremento (--) funciona de manera similar. Observe que este ejemplo sólo contiene una clase, en donde el método main realiza todo el trabajo de ésta. En éste y en el capítulo 3, usted ha visto ejemplos que consisten en dos clases: una clase contiene los métodos que realizan tareas útiles, y la otra contiene el método main, que crea un objeto de la otra clase y hace llamadas a sus métodos. En este ejemplo simplemente queremos mostrarle la mecánica del operador ++, por lo que sólo usaremos una declaración de clase que contiene el método main. Algunas veces, cuando no tenga sentido tratar de crear una clase reutilizable para demostrar un concepto simple, utilizaremos un ejemplo "mecánico" contenido completamente dentro del método main de una sola clase.

La línea 11 inicializa la variable c con 5, y la línea 12 imprime el valor inicial de c. La línea 13 imprime el valor de la expresión c++. Esta expresión postincrementa la variable c, por lo que se imprime el valor original de c (5), y después el valor de c se incrementa (a 6). Por ende, la línea 13 imprime el valor inicial de c (5) otra vez. La línea 14 imprime el nuevo valor de c (6) para demostrar que, sin duda, se incrementó el valor de la variable en la línea 13.

La línea 19 restablece el valor de c a 5, y la línea 20 imprime el valor de c. La línea 21 imprime el valor de la expresión ++c. Esta expresión preincrementa a c, por lo que su valor se incrementa y después se imprime el nuevo valor (6). La línea 22 imprime el valor de c otra vez, para mostrar que sigue siendo 6 después de que se ejecuta la línea 21.

Los operadores de asignación compuestos aritméticos y los operadores de incremento y decremento pueden utilizarse para simplificar las instrucciones de los programas. Por ejemplo, las tres instrucciones de asignación de la figura 4.12 (líneas 27, 29 y 32)

```
aprobados = aprobados + 1;
reprobados = reprobados + 1;
contadorEstudiantes = contadorEstudiantes + 1;
```

pueden escribirse en forma más concisa con operadores de asignación compuestos, de la siguiente manera:

```
aprobados += 1;
reprobados += 1;
contadorEstudiantes += 1;
```

con operadores de preincremento de la siguiente forma:

```
++aprobados;
++reprobados;
++contadorEstudiantes;
```

o con operadores de postincremento de la siguiente forma:

```
aprobados++;
reprobados++;
contadorEstudiantes++;
```

```
// Fig. 4.16: Incremento.java
// Operadores de preincremento y postincremento.

public class Incremento
{
   public static void main( String args[] )
   {
      int c;

      // demuestra el operador de preincremento
      c = 5; // asigna 5 a c
      System.out.println( c );   // imprime 5
      System.out.println( c++ ); // imprime 5, después postincrementa
      System.out.println( c );   // imprime 6

      System.out.println(); // omite una línea

      // demuestra el operador de postincremento
      c = 5; // asigna 5 a c
      System.out.println( c );   // imprime 5
      System.out.println( ++c ); // preincrementa y después imprime 6
      System.out.println( c );   // imprime 6

   } // fin de main

} // fin de la clase Incremento
```

```
5
5
6

5
6
6
```

Figura 4.16 | Preincrementar y postincrementar.

Al incrementar o decrementar una variable que se encuentre en una instrucción por sí sola, las formas preincremento y postincremento tienen el mismo efecto, al igual que las formas predecremento y postdecremento. Solamente cuando una variable aparece en el contexto de una expresión más grande es cuando los operadores preincremento y postdecremento tienen distintos efectos (y lo mismo se aplica a los operadores de predecremento y postdecremento).



Error común de programación 4.9

Tratar de usar el operador de incremento o decremento en una expresión a la que no se le pueda asignar un valor es un error de sintaxis. Por ejemplo, escribir `++(x + 1)` *es un error de sintaxis, ya que* `(x + 1)` *no es una variable.*

La figura 4.17 muestra la precedencia y la asociatividad de los operadores que se han presentado hasta este punto. Los operadores se muestran de arriba a abajo, en orden descendente de precedencia. La segunda columna describe la asociatividad de los operadores en cada nivel de precedencia. El operador condicional (`?:`), los operadores unarios de incremento (`++`), decremento (`--`), suma (`+`) y resta (`-`), los operadores de conversión de tipo y los operadores de asignación `=`, `+=`, `-=`, `*=`, `/=` y `%=` se asocian de derecha a izquierda. Todos los demás operadores en la tabla de precedencia de operadores de la figura 4.17 se asocian de izquierda a derecha. La tercera columna enlista el tipo de cada grupo de operadores.

Operadores						Asociatividad	Tipo
++	--					derecha a izquierda	postfijo unario
++	--	+	-	(*tipo*)		derecha a izquierda	prefijo unario
*	/	%				izquierda a derecha	multiplicativo
+	-					izquierda a derecha	aditivo
<	<=	>	>=			izquierda a derecha	relacional
==	!=					izquierda a derecha	igualdad
?:						derecha a izquierda	condicional
=	+=	-=	*=	/=	%=	derecha a izquierda	asignación

Figura 4.17 | Precedencia y asociatividad de los operadores vistos hasta ahora.

4.13 Tipos primitivos

La tabla del apéndice D, Tipos primitivos, enlista los ocho tipos primitivos en Java. Al igual que sus lenguajes antecesores C y C++, Java requiere que todas las variables tengan un tipo. Es por esta razón que Java se conoce como un **lenguaje fuertemente tipificado**.

En C y C++, los programadores frecuentemente tienen que escribir versiones independientes de los programas, ya que no se garantiza que los tipos primitivos sean idénticos de computadora en computadora. Por ejemplo, un valor `int` en un equipo podría representarse mediante 16 bits (2 bytes) de memoria, mientras que un valor `int` en otro equipo podría representarse mediante 32 bits (4 bytes) de memoria. En Java, los valores `int` siempre son de 32 bits (4 bytes).



Tip de portabilidad 4.1

A diferencia de C y C++, los tipos primitivos en Java son portables en todas las plataformas con soporte para Java.

Cada uno de los tipos del apéndice D se enlista con su tamaño en bits (hay ocho bits en un byte) y su rango de valores. Como los diseñadores de Java desean que sea lo más portable posible, utilizan estándares reconocidos internacionalmente tanto para los formatos de caracteres (Unicode; para más información, visite `www.unicode.org`) como para los números de punto flotante (IEEE 754; para más información, visite `grouper.ieee.org/groups/754/`).

En la sección 3.5 vimos que a las variables de tipos primitivos que se declaran fuera de un método, como campos de una clase, se les asignan valores predeterminados, a menos que se inicialicen en forma explícita. Las variables de los tipos `char`, `byte`, `short`, `int`, `long`, `float` y `double` reciben el valor `0` de manera predeterminada. Las variables de tipo `boolean` reciben el valor `false` de manera predeterminada. Las variables de instancia de tipo por referencia se inicializan de manera predeterminada con el valor `null`.

4.14 (Opcional) Ejemplo práctico de GUI y gráficos: creación de dibujos simples

Una de las características interesantes de Java es su soporte para gráficos, el cual permite a los programadores mejorar visualmente sus aplicaciones. Esta sección presenta una de las capacidades gráficas de Java: dibujar líneas. También cubre los aspectos básicos acerca de cómo crear una ventana para mostrar un dibujo en la pantalla de la computadora.

Para dibujar en Java, debe comprender su **sistema de coordenadas** (figura 4.18), un esquema para identificar cada uno de los puntos en la pantalla. De manera predeterminada, la esquina superior izquierda de un componente de la GUI tiene las coordenadas (0, 0). Un par de coordenadas está compuesto por una **coordenada *x*** (la

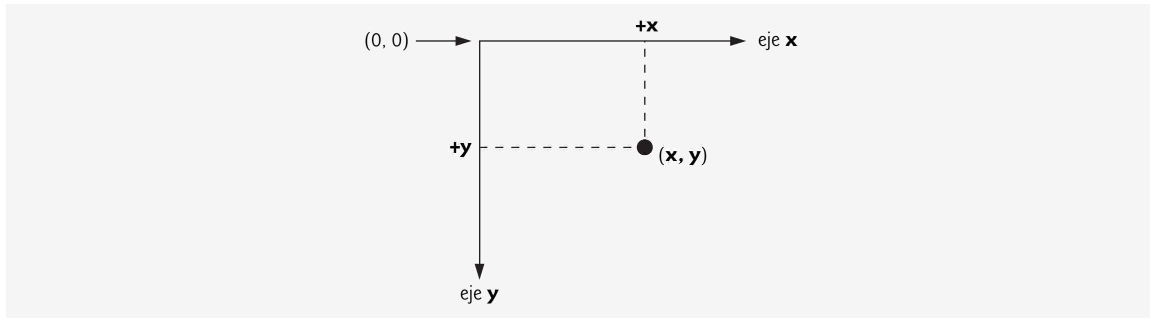


Figura 4.18 | Sistema de coordenadas de Java. Las unidades se miden en píxeles.

coordenada horizontal) y una **coordenada *y*** (la **coordenada vertical**). La coordenada *x* es la ubicación horizontal que se desplaza de izquierda a derecha. La coordenada *y* es la ubicación vertical que se desplaza de arriba hacia abajo. El **eje *x*** describe cada una de las coordenadas horizontales, y el **eje *y*** describe cada una de las coordenadas verticales.

Las coordenadas indican en dónde deben mostrarse los gráficos en una pantalla. Las unidades de las coordenadas se miden en **píxeles**. Un píxel es la unidad de resolución más pequeña de una pantalla. (El término píxel significa "elemento de imagen").

Nuestra primera aplicación de dibujo simplemente dibuja dos líneas. La clase `PanelDibujo` (figura 4.19) realiza el dibujo en sí, mientras que la clase `PruebaPanelDibujo` (figura 4.20) crea una ventana para mostrar el dibujo. En la clase `PanelDibujo`, las instrucciones `import` de las líneas 3 y 4 nos permiten utilizar la clase **`Graphics`** (del paquete `java.awt`), que proporciona varios métodos para dibujar texto y figuras en la pantalla, y la clase **`JPanel`** (del paquete `javax.swing`), que proporciona un área en la que podemos dibujar.

La línea 6 utiliza la palabra clave **`extends`** para indicar que la clase `PanelDibujo` es un tipo mejorado de `JPanel`. La palabra clave `extends` representa algo que se denomina relación de herencia, en la cual nuestra nueva clase `PanelDibujo` empieza con los miembros existentes (datos y métodos) de la clase `JPanel`. La clase de la cual

```
1  // Fig. 4.19: PanelDibujo.java
2  // Uso de drawLine para conectar las esquinas de un panel.
3  import java.awt.Graphics;
4  import javax.swing.JPanel;
5
6  public class PanelDibujo extends JPanel
7  {
8     // dibuja una x desde las esquinas del panel
9     public void paintComponent( Graphics g )
10    {
11       // llama a paintComponent para asegurar que el panel se muestre correctamente
12       super.paintComponent( g );
13
14       int anchura = getWidth(); // anchura total
15       int altura = getHeight(); // altura total
16
17       // dibuja una línea de la esquina superior izquierda a la esquina inferior derecha
18       g.drawLine( 0, 0, anchura, altura );
19
20       // dibuja una línea de la esquina inferior izquierda a la esquina superior derecha
21       g.drawLine( 0, altura, anchura, 0 );
22    } // fin del método paintComponent
23 } // fin de la clase PanelDibujo
```

Figura 4.19 | Uso de `drawLine` para conectar las esquinas de un panel.

`PanelDibujo` **hereda**, `JPanel`, aparece a la derecha de la palabra clave `extends`. En esta relación de herencia, a `JPanel` se le conoce como la **superclase** y `PanelDibujo` es la **subclase**. Esto produce una clase `PanelDibujo` que tiene los atributos (datos) y comportamientos (métodos) de la clase `JPanel`, así como las nuevas características que agregaremos en nuestra declaración de la clase `PanelDibujo`; específicamente, la habilidad de dibujar dos líneas a lo largo de las diagonales del panel. En el capítulo 9 explicaremos detalladamente el concepto de herencia.

Todo `JPanel`, incluyendo nuestro `PanelDibujo`, tiene un método **`paintComponent`** (líneas 9 a 22), que el sistema llama automáticamente cada vez que necesita mostrar el objeto `JPanel`. El método `paintComponent` debe declararse como se muestra en la línea 9; de no ser así, el sistema no llamará al método. Este método se llama cuando se muestra un objeto `JPanel` por primera vez en la pantalla, cuando una ventana en la pantalla lo cubre y después lo descubre, y cuando la ventana en la que aparece cambia su tamaño. El método `paintComponent` requiere un argumento, un objeto `Graphics`, que el sistema proporciona por usted cuando llama a `paintComponent`.

La primera instrucción en cualquier método `paintComponent` que cree debe ser siempre:

```
super.paintComponent( g );
```

la cual asegura que el panel se despliegue apropiadamente en la pantalla, antes de empezar a dibujar en él. A continuación, las líneas 14 y 15 llaman a dos métodos que la clase `PanelDibujo` hereda de la clase `JPanel`. Como `PanelDibujo` extiende a `JPanel`, `PanelDibujo` puede usar cualquier método `public` que esté declarado en `JPanel`. Los métodos **`getWidth`** y **`getHeight`** devuelven la anchura y la altura del objeto `JPanel`, respectivamente. Las líneas 14 y 15 almacenan estos valores en las variables locales `anchura` y `altura`. Por último, las líneas 18 y 21 utilizan la referencia `g` de la clase `Graphics` para llamar al método **`drawLine`**, y que dibuje las dos líneas. Los primeros dos argumentos son las coordenadas *x* y *y* para uno de los puntos finales de la línea, y los últimos dos argumentos son las coordenadas para el otro punto final. Si cambia de tamaño la ventana, las líneas se escalaran de manera acorde, ya que los argumentos se basan en la anchura y la altura del panel. Al cambiar el tamaño de la ventana en esta aplicación, el sistema llama a `paintComponent` para volver a dibujar el contenido de `PanelDibujo`.

Para mostrar el `PanelDibujo` en la pantalla, debemos colocarlo en una ventana. Usted debe crear una ventana con un objeto de la clase **`JFrame`**. En `PruebaPanelDibujo.java` (figura 4.20), la línea 3 importa la clase `JFrame` del paquete `javax.swing`. La línea 10 en el método `main` de la clase `PruebaPanelDibujo` crea una instancia de la clase `PanelDibujo`, la cual contiene nuestro dibujo, y la línea 13 crea un nuevo objeto `JFrame` que puede contener y mostrar nuestro panel. La línea 16 llama al método **`setDefaultCloseOperation`** con el argumento `JFrame.EXIT_ON_CLOSE`, para indicar que la aplicación debe terminar cuando el usuario cierre la ventana. La línea 18 utiliza el método **`add`** de `JFrame` para adjuntar el objeto `PanelDibujo`, que contiene nuestro dibujo, al objeto `JFrame`. La línea 19 establece el tamaño del objeto `JFrame`. El método **`setSize`** recibe dos parámetros: la anchura del objeto `JFrame` y la altura. Por último, la línea 20 muestra el objeto `JFrame`. Cuando se muestra este objeto, se hace la llamada al método `paintComponent` de `PanelDibujo` (líneas 9 a 22 de la figura 4.19) y se dibujan las dos líneas (vea los resultados de ejemplo de la figura 4.20). Cambie el tamaño de la ventana, para que vea que las líneas siempre se dibujan con base en la anchura y altura actuales de la ventana.

Ejercicios del ejemplo práctico de GUI y gráficos

4.1 Utilizar ciclos e instrucciones de control para dibujar líneas puede producir muchos diseños interesantes.

a) Cree el diseño que se muestra en la captura de pantalla izquierda de la figura 4.21. Este diseño dibuja líneas que parten desde la esquina superior izquierda, y se despliegan hasta cubrir la mitad superior izquierda del panel. Un método es dividir la anchura y la altura en un número equivalente de pasos (nosotros descubrimos que 15 pasos es una buena cantidad). El primer punto final de una línea siempre estará en la esquina superior izquierda (0,0). El segundo punto final puede encontrarse partiendo desde la esquina inferior izquierda, y avanzando un paso vertical hacia arriba, y un paso horizontal hacia la derecha. Dibuje una línea entre los dos puntos finales. Continúe avanzando hacia arriba y a la derecha, para encontrar cada punto final sucesivo. La figura deberá escalarse apropiadamente, a medida que se cambie el tamaño de la ventana.

b) Modifique su respuesta en la parte (a) para hacer que las líneas se desplieguen a partir de las cuatro esquinas, como se muestra en la captura de pantalla derecha de la figura 4.21. Las líneas de esquinas opuestas deberán intersecarse a lo largo de la parte media.

4.2 La figura 4.22 muestra dos diseños adicionales, creados mediante el uso de ciclos `while` y `drawLine`.

a) Cree el diseño de la captura de pantalla izquierda de la figura 4.22. Empiece por dividir cada flanco en un número equivalente de incrementos (elegimos 15 de nuevo). La primera línea empieza en la esquina superior izquierda y termina un paso a la derecha, en el flanco inferior. Para cada línea sucesiva, avance hacia abajo un incremento

```java
// Fig. 4.20: PruebaPanelDibujo.java
// Aplicación que muestra un PanelDibujo.
import javax.swing.JFrame;

public class PruebaPanelDibujo
{
   public static void main( String args[] )
   {
      // crea un panel que contiene nuestro dibujo
      PanelDibujo panel = new PanelDibujo();

      // crea un nuevo marco para contener el panel
      JFrame aplicacion = new JFrame();

      // establece el marco para salir cuando se cierre
      aplicacion.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );

      aplicacion.add( panel ); // agrega el panel al marco
      aplicacion.setSize( 250, 250 ); // establece el tamaño del marco
      aplicacion.setVisible( true ); // hace que el marco sea visible
   } // fin de main
} // fin de la clase PruebaPanelDibujo
```

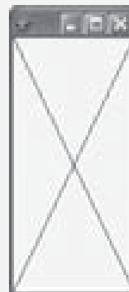



Figura 4.20 | Creación de un objeto `JFrame` para mostrar el objeto `PanelDibujo`.

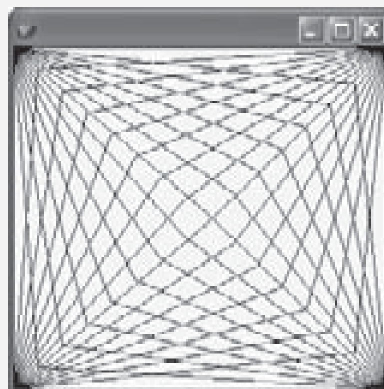



Figura 4.21 | Despliegue de líneas desde una esquina.

en el flanco izquierdo, y un incremento a la derecha en el flanco inferior. Continúe dibujando líneas hasta llegar a la esquina inferior derecha. La figura deberá escalarse a medida que se cambie el tamaño de la ventana, de manera que los puntos finales siempre toquen los flancos.

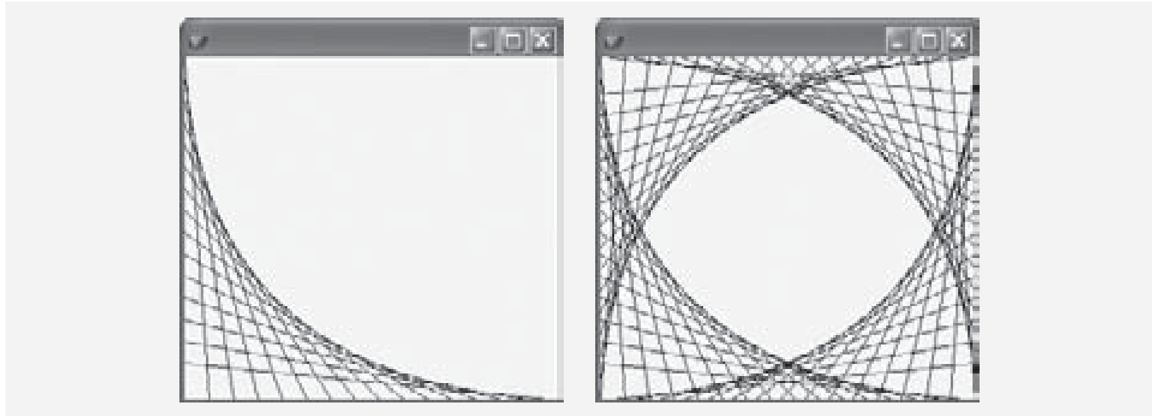


Figura 4.22 | Arte lineal con ciclos y `drawLine`.

b) Modifique su respuesta en la parte (a) para reflejar el diseño en las cuatro esquinas, como se muestra en la captura de pantalla derecha de la figura 4.22.

4.15 (Opcional) Ejemplo práctico de Ingeniería de Software: identificación de los atributos de las clases

En la sección 3.10 empezamos la primera etapa de un diseño orientado a objetos (DOO) para nuestro sistema ATM: analizar el documento de requerimientos e identificar las clases necesarias para implementar el sistema. Enlistamos los sustantivos y las frases nominales en el documento de requerimientos e identificamos una clase separada para cada uno de ellos, que desempeña un papel importante en el sistema ATM. Después modelamos las clases y sus relaciones en un diagrama de clases de UML (figura 3.24). Las clases tienen atributos (datos) y operaciones (comportamientos). En los programas en Java, los atributos de las clases se implementan como campos, y las operaciones de las clases se implementan como métodos. En esta sección determinaremos muchos de los atributos necesarios en el sistema ATM. En el capítulo 5 examinaremos cómo esos atributos representan el estado de un objeto. En el capítulo 6 determinaremos las operaciones de las clases.

Identificación de los atributos

Considere los atributos de algunos objetos reales: los atributos de una persona incluyen su altura, peso y si es zurdo, diestro o ambidiestro. Los atributos de un radio incluyen la estación, el volumen y si está en AM o FM. Los atributos de un automóvil incluyen las lecturas de su velocímetro y odómetro, la cantidad de gasolina en su tanque y la velocidad de marcha en la que se encuentra. Los atributos de una computadora personal incluyen su fabricante (por ejemplo, Dell, Sun, Apple o IBM), el tipo de pantalla (por ejemplo, LCD o CRT), el tamaño de su memoria principal y el de su disco duro.

Podemos identificar muchos atributos de las clases en nuestro sistema, analizando las palabras y frases descriptivas en el documento de requerimientos. Para cada palabra o frase que descubramos desempeña un rol importante en el sistema ATM, creamos un atributo y lo asignamos a una o más de las clases identificadas en la sección 3.10. También creamos atributos para representar los datos adicionales que pueda necesitar una clase, ya que dichas necesidades se van aclarando a lo largo del proceso de diseño.

La figura 4.23 lista las palabras o frases del documento de requerimientos que describen a cada una de las clases. Para formar esta lista, leemos el documento de requerimientos e identificamos cualquier palabra o frase que haga referencia a las características de las clases en el sistema. Por ejemplo, el documento de requerimientos describe los pasos que se llevan a cabo para obtener un "monto de retiro", por lo que listamos "monto" enseguida de la clase `Retiro`.

La figura 4.23 nos conduce a crear un atributo de la clase `ATM`. Esta clase mantiene información acerca del estado del ATM. La frase "el usuario es autenticado" describe un estado del ATM (en la sección 5.11 hablaremos con detalle sobre los estados), por lo que incluimos `usuarioAutenticado` como un **atributo `Boolean`** (es decir, un atributo que tiene un valor de `true` o `false`) en la clase `ATM`. Observe que el atributo tipo `Boolean` en UML

Clase	Palabras y frases descriptivas
ATM	el usuario es autenticado
SolicitudSaldo	número de cuenta
Retiro	número de cuenta monto
Deposito	número de cuenta monto
BaseDatosBanco	[no hay palabras o frases descriptivas]
Cuenta	número de cuenta NIP saldo
Pantalla	[no hay palabras o frases descriptivas]
Teclado	[no hay palabras o frases descriptivas]
DispensadorEfectivo	empieza cada día cargado con 500 billetes de $20
RanuraDeposito	[no hay palabras o frases descriptivas]

Figura 4.23 | Palabras y frases descriptivas del documento de requerimientos del ATM.

es equivalente al tipo `boolean` en Java. Este atributo indica si el ATM autenticó con éxito al usuario actual o no; `usuarioAutenticado` debe ser `true` para que el sistema permita al usuario realizar transacciones y acceder a la información de la cuenta. Este atributo nos ayuda a cerciorarnos de la seguridad de los datos en el sistema.

Las clases `SolicitudSaldo`, `Retiro` y `Deposito` comparten un atributo. Cada transacción requiere un "número de cuenta" que corresponde a la cuenta del usuario que realiza la transacción. Asignamos el atributo entero `numeroCuenta` a cada clase de transacción para identificar la cuenta a la que se aplica un objeto de la clase.

Las palabras y frases descriptivas en el documento de requerimientos también sugieren ciertas diferencias en los atributos requeridos por cada clase de transacción. El documento de requerimientos indica que para retirar efectivo o depositar fondos, los usuarios deben introducir un "monto" específico de dinero para retirar o depositar, respectivamente. Por ende, asignamos a las clases `Retiro` y `Deposito` un atributo llamado `monto` para almacenar el valor suministrado por el usuario. Los montos de dinero relacionados con un retiro y un depósito son características que definen estas transacciones, que el sistema requiere para que se lleven a cabo. Sin embargo, la clase `SolicitudSaldo` no necesita datos adicionales para realizar su tarea; sólo requiere un número de cuenta para indicar la cuenta cuyo saldo hay que obtener.

La clase `Cuenta` tiene varios atributos. El documento de requerimientos establece que cada cuenta de banco tiene un "número de cuenta" y un "NIP", que el sistema utiliza para identificar las cuentas y autentificar a los usuarios. A la clase `Cuenta` le asignamos dos atributos enteros: `numeroCuenta` y `nip`. El documento de requerimientos también especifica que una cuenta debe mantener un "saldo" del monto de dinero que hay en la cuenta, y que el dinero que el usuario deposita no estará disponible para su retiro sino hasta que el banco verifique la cantidad de efectivo en el sobre de depósito y cualquier cheque que contenga. Sin embargo, una cuenta debe registrar de todas formas el monto de dinero que deposita un usuario. Por lo tanto, decidimos que una cuenta debe representar un saldo utilizando dos atributos: `saldoDisponible` y `saldoTotal`. El atributo `saldoDisponible` rastrea el monto de dinero que un usuario puede retirar de la cuenta. El atributo `saldoTotal` se refiere al monto total de dinero que el usuario tiene "en depósito" (es decir, el monto de dinero disponible, más el monto de depósitos en efectivo o la cantidad de cheques esperando a ser verificados). Por ejemplo, suponga que un usuario del ATM deposita $50.00 en efectivo, en una cuenta vacía. El atributo `saldoTotal` se incrementaría a $50.00 para registrar el depósito, pero el `saldoDisponible` permanecería en $0. [*Nota:* estamos suponiendo que el banco actualiza el atributo `saldoDisponible` de una `Cuenta` poco después de que se realiza la transacción del ATM, en respuesta a la confirmación de que se encontró un monto equivalente a $50.00 en efectivo o cheques en el sobre de depósito. Asumimos que esta actualización se realiza a través de una transacción que realiza el empleado del banco mediante

el uso de un sistema bancario distinto al del ATM. Por ende, no hablaremos sobre esta transacción en nuestro ejemplo práctico].

La clase `DispensadorEfectivo` tiene un atributo. El documento de requerimientos establece que el dispensador de efectivo "empieza cada día cargado con 500 billetes de $20". El dispensador de efectivo debe llevar el registro del número de billetes que contiene para determinar si hay suficiente efectivo disponible para satisfacer la demanda de los retiros. Asignamos a la clase `DispensadorEfectivo` el atributo entero `conteo`, el cual se establece al principio en `500`.

Para los verdaderos problemas en la industria, no existe garantía alguna de que el documento de requerimientos será lo suficientemente robusto y preciso como para que el diseñador de sistemas orientados a objetos determine todos los atributos, o inclusive todas las clases. La necesidad de clases, atributos y comportamientos adicionales puede irse aclarando a medida que avance el proceso de diseño. A medida que progresemos a través de este ejemplo práctico, nosotros también seguiremos agregando, modificando y eliminando información acerca de las clases en nuestro sistema.

Modelado de los atributos

El diagrama de clases de la figura 4.24 enlista algunos de los atributos para las clases en nuestro sistema; las palabras y frases descriptivas en la figura 4.23 nos llevan a identificar estos atributos. Por cuestión de simpleza, la figura 4.24 no muestra las asociaciones entre las clases; en la figura 3.24 mostramos estas asociaciones. Ésta es una práctica común de los diseñadores de sistemas, a la hora de desarrollar los diseños. En la sección 3.10 vimos que en UML, los atributos de una clase se colocan en el compartimiento intermedio del rectángulo de la clase. Listamos el nombre de cada atributo y su tipo, separados por un signo de dos puntos (`:`), seguido en algunos casos de un signo de igual (=) y de un valor inicial.

Considere el atributo `usuarioAutenticado` de la clase `ATM`:

```
usuarioAutenticado : Boolean = false
```

La declaración de este atributo contiene tres piezas de información acerca del atributo. El **nombre del atributo** es `usuarioAutenticado`. El **tipo del atributo** es `Boolean`. En Java, un atributo puede representarse mediante un tipo primitivo, como `boolean`, `int` o `double`, o por un tipo de referencia como una clase (como vimos en el capítulo 3). Hemos optado por modelar sólo los atributos de tipo primitivo en la figura 4.24; en breve hablaremos sobre el razonamiento detrás de esta decisión. [*Nota:* los tipos de los atributos en la figura 4.24 están en notación de UML. Asociaremos los tipos `Boolean`, `Integer` y `Double` en el diagrama de UML con los tipos primitivos `boolean`, `int` y `double` en Java, respectivamente].

También podemos indicar un valor inicial para un atributo. El atributo `usuarioAutenticado` en la clase `ATM` tiene un valor inicial de `false`. Esto indica que al principio el sistema no considera que el usuario está autenticado. Si no se especifica un valor inicial para un atributo, sólo se muestran su nombre y tipo (separados por dos puntos). Por ejemplo, el atributo `numeroCuenta` de la clase `SolicitudSaldo` es un entero. Aquí no mostramos un valor inicial, ya que el valor de este atributo es un número que todavía no conocemos. Este número se determinará en tiempo de ejecución, con base en el número de cuenta introducido por el usuario actual del ATM.

La figura 4.24 no incluye atributos para las clases `Pantalla`, `Teclado` y `RanuraDeposito`. Éstos son componentes importantes de nuestro sistema, para los cuales nuestro proceso de diseño aún no ha revelado ningún atributo. No obstante, tal vez descubramos algunos en las fases restantes de diseño, o cuando implementemos estas clases en Java. Esto es perfectamente normal.



Observación de ingeniería de software 4.6

En las primeras fases del proceso de diseño, a menudo las clases carecen de atributos (y operaciones). Sin embargo, esas clases no deben eliminarse, ya que los atributos (y las operaciones) pueden hacerse evidentes en las fases posteriores de diseño e implementación.

Observe que la figura 4.24 tampoco incluye atributos para la clase `BaseDatosBanco`. En el capítulo 3 vimos que en Java, los atributos pueden representarse mediante los tipos primitivos o los tipos por referencia. Hemos optado por incluir sólo los atributos de tipo primitivo en el diagrama de clases de la figura 4.24 (y en los diagramas de clases similares a lo largo del ejemplo práctico). Un atributo de tipo por referencia se modela con más claridad como una asociación (en particular, una composición) entre la clase que contiene la referencia y la clase del objeto al que apunta la referencia. Por ejemplo, el diagrama de clases de la figura 3.24 indica que

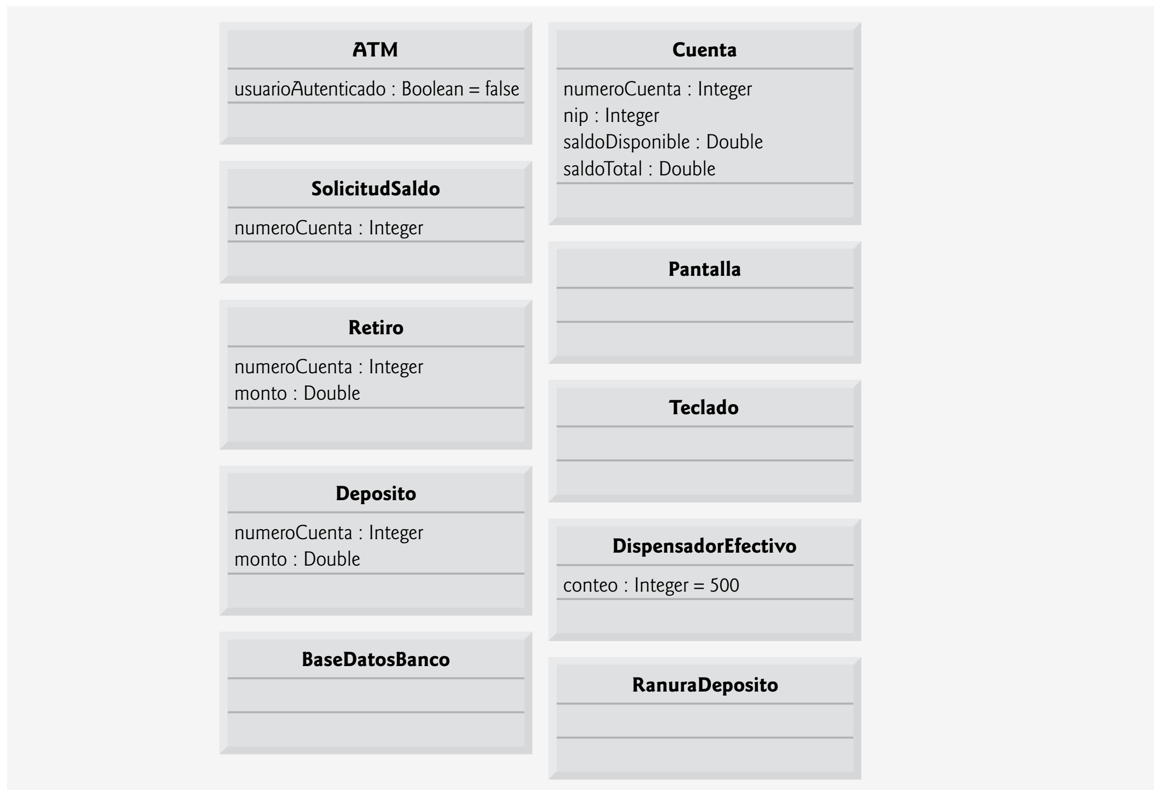


Figura 4.24 | Clases con atributos.

la clase `BaseDatosBanco` participa en una relación de composición con cero o más objetos `Cuenta`. De esta composición podemos determinar que, cuando implementemos el sistema ATM en Java, tendremos que crear un atributo de la clase `BaseDatosBanco` para almacenar cero o más objetos `Cuenta`. De manera similar, podemos determinar los atributos de tipo por referencia de la clase `ATM` que correspondan a sus relaciones de composición con las clases `Pantalla`, `Teclado`, `DispensadorEfectivo` y `RanuraDeposito`. Estos atributos basados en composiciones serían redundantes si los modeláramos en la figura 4.24, ya que las composiciones modeladas en la figura 3.24 transmiten de antemano el hecho de que la base de datos contiene información acerca de cero o más cuentas, y que un ATM está compuesto por una pantalla, un teclado, un dispensador de efectivo y una ranura para depósitos. Por lo general, los desarrolladores de software modelan estas relaciones de todo/parte como asociaciones de composición, en vez de modelarlas como atributos requeridos para implementar las relaciones.

El diagrama de clases de la figura 4.24 proporciona una base sólida para la estructura de nuestro modelo, pero no está completo. En la sección 5.11 identificaremos los estados y las actividades de los objetos en el modelo, y en la sección 6.14 identificaremos las operaciones que realizan los objetos. A medida que presentemos más acerca de UML y del diseño orientado a objetos, continuaremos reforzando la estructura de nuestro modelo.

Ejercicios de autoevaluación del Ejemplo práctico de Ingeniería de Software

4.1 Por lo general, identificamos los atributos de las clases en nuestro sistema mediante el análisis de ____________ en el documento de requerimientos.

a) Los sustantivos y las frases nominales.
b) Las palabras y frases descriptivas.
c) Los verbos y las frases verbales.
d) Todo lo anterior.

4.2 ¿Cuál de los siguientes no es un atributo de un aeroplano?

a) Longitud.
b) Envergadura.

c) Volar.
d) Número de asientos.

4.3 Describa el significado de la siguiente declaración de un atributo de la clase `DispensadorEfectivo` en el diagrama de clases de la figura 4.24:

```
conteo : Integer = 500
```

Respuestas a los ejercicios de autoevaluación del Ejemplo práctico de Ingeniería de Software

4.1 b.

4.2 c. Volar es una operación o comportamiento de un aeroplano, no un atributo.

4.3 Esta declaración indica que el atributo `conteo` es de tipo `Integer`, con un valor inicial de `500`. Este atributo lleva la cuenta del número de billetes disponibles en el `DispensadorEfectivo`, en cualquier momento dado.

4.16 Conclusión

Este capítulo presentó las estrategias básicas de solución de problemas, que los programadores utilizan para crear clases y desarrollar métodos para estas clases. Demostramos cómo construir un algoritmo (es decir, una metodología para resolver un problema), y después cómo refinar el algoritmo a través de diversas fases de desarrollo de seudocódigo, lo cual produce código en Java que puede ejecutarse como parte de un método. El capítulo demostró cómo utilizar el método de refinamiento de arriba a abajo, paso a paso, para planear las acciones específicas que debe realizar un método, y el orden en el que debe realizar estas acciones.

Sólo se requieren tres tipos de estructuras de control (secuencia, selección y repetición) para desarrollar cualquier algoritmo para solucionar un problema. Específicamente, en este capítulo demostramos el uso de la instrucción de selección simple `if`, la instrucción de selección doble `if...else` y la instrucción de repetición `while`. Estas instrucciones son algunos de los bloques de construcción que se utilizan para construir soluciones para muchos problemas. Utilizamos el apilamiento de instrucciones de control para calcular el total y el promedio de un conjunto de calificaciones de estudiantes, mediante la repetición controlada por un contador y controlada por un centinela, y utilizamos el anidamiento de instrucciones de control para analizar y tomar decisiones con base en un conjunto de resultados de un examen. Presentamos los operadores de asignación compuestos de Java, así como sus operadores de incremento y decremento. Por último, hablamos sobre los tipos primitivos disponibles para los programadores de Java. En el capítulo 5, Instrucciones de control: parte 2, continuaremos nuestra discusión acerca de las instrucciones de control, en donde presentaremos las instrucciones `for`, `do...while` y `switch`.

Resumen

Sección 4.1 Introducción

- Antes de escribir un programa para resolver un problema, debe tener una comprensión detallada acerca del problema y una metodología cuidadosamente planeada para resolverlo. También debe comprender los bloques de construcción disponibles, y emplear las técnicas probadas para construir programas.

Sección 4.2 Algoritmos

- Cualquier problema de cómputo puede resolverse mediante la ejecución de una serie de acciones, en un orden específico.
- Un procedimiento para resolver un problema, en términos de las acciones a ejecutar y el orden en el que se ejecutan, se denomina algoritmo.
- El proceso de especificar el orden en el que se ejecutan las instrucciones en un programa se denomina control del programa.

Sección 4.3 Seudocódigo

- El seudocódigo es un lenguaje informal, que ayuda a los programadores a desarrollar algoritmos sin tener que preocuparse por los estrictos detalles de la sintaxis del lenguaje Java.
- El seudocódigo es similar al lenguaje cotidiano; es conveniente y amigable para el usuario, pero no es un verdadero lenguaje de programación de computadoras.

- El seudocódigo ayuda al programador a "idear" un programa antes de intentar escribirlo en un lenguaje de programación.
- El seudocódigo cuidadosamente preparado puede convertirse con facilidad en su correspondiente programa en Java.

Sección 4.4 Estructuras de control

- Por lo general, las instrucciones en un programa se ejecutan, una después de la otra, en el orden en el que están escritas. A este proceso se le conoce como ejecución secuencial.
- Varias instrucciones de Java permiten al programador especificar que la siguiente instrucción a ejecutar no es necesariamente la siguiente en la secuencia. A esto se le conoce como transferencia de control.
- Bohm y Jacopini demostraron que todos los programas podían escribirse en términos de sólo tres estructuras de control: la estructura de secuencia, la estructura de selección y la estructura de repetición.
- El término "estructuras de control" proviene del campo de las ciencias computacionales. La *Especificación del lenguaje Java* se refiere a las "estructuras de control" como "instrucciones de control".
- La estructura de secuencia está integrada en Java. A menos que se indique lo contrario, la computadora ejecuta las instrucciones de Java, una después de la otra, en el orden en el que están escritas; es decir, en secuencia.
- En cualquier parte en donde pueda colocarse una sola acción, pueden colocarse varias acciones en secuencia.
- Los diagramas de actividad forman parte de UML. Un diagrama de actividad modela el flujo de trabajo (también conocido como la actividad) de una parte de un sistema de software.
- Los diagramas de actividad se componen de símbolos de propósito especial, como los símbolos de estados de acción, rombos y pequeños círculos. Estos símbolos se conectan mediante flechas de transición, las cuales representan el flujo de la actividad.
- Los estados de acción representan las acciones a realizar. Cada estado de acción contiene una expresión de acción, la cual especifica una acción específica a realizar.
- Las flechas en un diagrama de actividad representan las transiciones, que indican el orden en el que ocurren las acciones representadas por los estados de acción.
- El círculo relleno que se encuentra en la parte superior de un diagrama de actividad representa el estado inicial de la actividad: el comienzo del flujo de trabajo antes de que el programa realice las acciones modeladas.
- El círculo sólido rodeado por una circunferencia, que aparece en la parte inferior del diagrama, representa el estado final: el término del flujo de trabajo después de que el programa realiza sus acciones.
- Los rectángulos con las esquinas superiores derechas dobladas se llaman notas en UML: comentarios que describen el propósito de los símbolos en el diagrama.
- Java tiene tres tipos de instrucciones de selección. La instrucción `if` realiza una acción si una condición es verdadera, o evita la acción si la condición es falsa. La instrucción `if...else` realiza una acción si una condición es verdadera, y realiza una acción distinta si la condición es falsa. La instrucción `switch` realiza una de varias acciones distintas, dependiendo del valor de una expresión.
- La instrucción `if` es una instrucción de selección simple, ya que selecciona o ignora una sola acción, o un solo grupo de acciones.
- La instrucción `if...else` se denomina instrucción de selección doble, ya que selecciona una de dos acciones distintas, o grupos de acciones.
- La instrucción `switch` se llama instrucción de selección múltiple, ya que selecciona una de varias acciones distintas, o grupos de acciones.
- Java cuenta con las instrucciones de repetición (ciclos) `while`, `do...while` y `for`, las cuales permiten a los programas ejecutar instrucciones en forma repetida, siempre y cuando una condición de continuación de ciclo siga siendo verdadera.
- Las instrucciones `while` y `for` realizan la(s) acción(es) en sus cuerpos, cero o más veces; si al principio la condición de continuación de ciclo es falsa, la(s) acción(es) no se ejecutará(n). La instrucción `do...while` lleva a cabo la(s) acción(es) que contiene en su cuerpo, una o más veces.
- Las palabras `if`, `else`, `switch`, `while`, `do` y `for` son palabras claves en Java. Las palabras clave no pueden utilizarse como identificadores, como los nombres de variables.
- Cada programa se forma mediante una combinación de todas las instrucciones de secuencia, selección y repetición que sean apropiadas para el algoritmo que implementa ese programa.
- Las instrucciones de control de una sola entrada/una sola salida facilitan la construcción de los programas; "adjuntamos" una instrucción de control a otra mediante la conexión del punto de salida de una al punto de entrada de la siguiente. A esto se le conoce como apilamiento de instrucciones de control.
- Sólo hay una forma alterna en la que pueden conectarse las instrucciones de control (anidamiento de instrucciones de control), en la cual una instrucción de control aparece dentro de otra instrucción de control.

Sección 4.5 Instrucción de selección simple `if`

- Los programas utilizan instrucciones de selección para elegir entre los cursos alternativos de acción.
- El diagrama de actividad de una instrucción `if` de selección simple contiene el rombo, o símbolo de decisión, el cual indica que se tomará una decisión. El flujo de trabajo continuará a lo largo de una ruta determinada por las condiciones de guardia asociadas al símbolo, que pueden ser verdaderas o falsas. Cada flecha de transición que emerge de un símbolo de decisión tiene una condición de guardia. Si una condición de guardia es verdadera, el flujo de trabajo entra al estado de acción al que apunta la flecha de transición.
- La instrucción `if` es una instrucción de control de una sola entrada/una sola salida.

Sección 4.6 Instrucción de selección doble `if...else`

- La instrucción `if` de selección simple realiza una acción indicada sólo cuando la condición es verdadera.
- La instrucción `if...else` de selección doble realiza una acción cuando la condición es verdadera, y otra acción distinta cuando la condición es falsa.
- El operador condicional (`?:`) puede usarse en lugar de una instrucción `if...else`. Éste es el único operador ternario de Java: recibe tres operandos. En conjunto, los operandos y el símbolo `?:` forman una expresión condicional.
- Un programa puede evaluar varios casos, colocando instrucciones `if...else` dentro de otras instrucciones `if...else`, para crear instrucciones `if...else` anidadas.
- El compilador de Java siempre asocia un `else` con el `if` que lo precede inmediatamente, a menos que se le indique otra cosa mediante la colocación de llaves (`{` y `}`). Este comportamiento puede conducir a lo que se conoce como el problema del `else` suelto.
- Por lo general, la instrucción `if` espera sólo una instrucción en su cuerpo. Para incluir varias instrucciones en el cuerpo de un `if` (o en el cuerpo de un `else` para una instrucción `if...else`), encierre las instrucciones entre llaves (`{` y `}`).
- A un conjunto de instrucciones contenidas dentro de un par de llaves se le llama bloque. Un bloque puede colocarse en cualquier parte de un programa, en donde se pueda colocar una sola instrucción.
- El compilador atrapa los errores de sintaxis.
- Un error lógico tiene su efecto en tiempo de ejecución. Un error lógico fatal hace que un programa falle y termine antes de tiempo. Un error lógico no fatal permite que un programa continúe ejecutándose, pero hace que el programa produzca resultados erróneos.
- Así como podemos colocar un bloque en cualquier parte en la que pueda colocarse una sola instrucción, también podemos usar una instrucción vacía, que se representa colocando un punto y coma (`;`) en donde normalmente estaría una instrucción.

Sección 4.7 Instrucción de repetición `while`

- La instrucción de repetición `while` permite al programador especificar que un programa debe repetir una acción, mientras cierta condición siga siendo verdadera.
- El símbolo de fusión de UML combina dos flujos de actividad en uno.
- Los símbolos de decisión y de fusión pueden diferenciarse en base al número de flechas de transición "entrantes" y "salientes". Un símbolo de decisión tiene una flecha de transición que apunta hacia el rombo, y dos o más flechas de transición que apuntan hacia fuera del rombo, para indicar las posibles transiciones desde ese punto. Cada flecha de transición que apunta hacia fuera de un símbolo de decisión tiene una condición de guardia. Un símbolo de fusión tiene dos o más flechas de transición que apuntan hacia el rombo, y sólo una flecha de transición que apunta hacia fuera del rombo, para indicar que se fusionarán varios flujos de actividad para continuar con la actividad. Ninguna de las flechas de transición asociadas con un símbolo de fusión tiene una condición de guardia.

Sección 4.8 Cómo formular algoritmos: repetición controlada por un contador

- La repetición controlada por un contador utiliza una variable llamada contador (o variable de control), para controlar el número de veces que se ejecuta un conjunto de instrucciones.
- A la repetición controlada por contador se le conoce comúnmente como repetición definida, ya que el número de repeticiones se conoce desde antes que empiece a ejecutarse el ciclo.
- Un total es una variable que se utiliza para acumular la suma de varios valores. Por lo general, las variables que se utilizan para almacenar totales se inicializan en cero antes de usarlas en un programa.
- La declaración de una variable local debe aparecer antes de usarla en el método en el que está declarada. Una variable local no puede utilizarse fuera del método en el que se declaró.
- Al dividir dos enteros se produce una división entera; la parte fraccionaria del cálculo se trunca.

Sección 4.9 Cómo formular algoritmos: repetición controlada por un centinela

- En la repetición controlada por un centinela, se utiliza un valor especial, conocido como valor centinela (valor de señal, valor de prueba o valor de bandera) para indicar el "fin de la entrada de datos".
- Debe elegirse un valor centinela que no pueda confundirse con un valor de entrada aceptable.
- El método de refinamiento de arriba a abajo, paso a paso, es esencial para el desarrollo de programas bien estructurados.
- Por lo general, la división entre cero es un error lógico que, si no se detecta, hace que el programa falle o que produzca resultados inválidos.
- Para realizar un cálculo de punto flotante con valores enteros, convierta uno de los enteros al tipo `double`. El uso de un operador de conversión de tipos de esta forma se denomina conversión explícita.
- Java sabe cómo evaluar sólo las expresiones aritméticas en las que los tipos de los operandos son idénticos. Para asegurar que los operandos sean del mismo tipo, Java realiza una operación conocida como promoción (o conversión implícita) sobre los operandos seleccionados. En una expresión que contiene valores de los tipos `int` y `double`, los valores `int` se promueven a valores `double` para usarlos en la expresión.
- Hay operadores de conversión de tipos disponibles para cualquier tipo. El operador de conversión de tipos se forma mediante la colocación de paréntesis alrededor del nombre de un tipo. Este operador es unario.

Sección 4.11 Operadores de asignación compuestos

- Java cuenta con varios operadores de asignación compuestos para abreviar las expresiones de asignación. Cualquier instrucción de la forma

  variable = *variable operador expresión*;

  en donde *operador* es uno de los operadores binarios `+`, `-`, `*`, `/` o `%`, puede escribirse en la forma

  variable operador= *expresión*;

- El operador `+=` suma el valor de la expresión que está a la derecha del operador, con el valor de la variable que está a la izquierda del operador, y almacena el resultado en la variable que está a la izquierda del operador.

Sección 4.12 Operadores de incremento y decremento

- Java cuenta con dos operadores unarios para sumar 1, o restar 1, al valor de una variable numérica. Éstos son el operador de incremento unario, `++`, y el operador de decremento unario, `--`.
- Un operador de incremento o decremento que se coloca antes de una variable es el operador de preincremento o predecremento, respectivamente. Un operador de incremento o decremento que se coloca después de una variable es el operador de postincremento o postdecremento, respectivamente.
- El proceso de usar el operador de preincremento o predecremento para sumar o restar 1 se conoce como preincrementar o predecrementar, respectivamente.
- Al preincrementar o predecrementar una variable, ésta se incrementa o decrementa por 1, y después se utiliza el nuevo valor de la variable en la expresión en la que aparece.
- El proceso de usar el operador de postincremento o postdecremento para sumar o restar 1 se conoce como postincrementar o postdecrementar, respectivamente.
- Al postincrementar o postdecrementar una variable, el valor actual de ésta se utiliza en la expresión en la que aparece, y después el valor de la variable se incrementa o decrementa por 1.
- Cuando se incrementa o decrementa una variable en una instrucción por sí sola, las formas de preincremento y postincremento tienen el mismo efecto, y las formas de predecremento y postdecremento tienen el mismo efecto.

Sección 4.13 Tipos primitivos

- Java requiere que todas las variables tengan un tipo. Por ende, Java se conoce como un lenguaje fuertemente tipificado.
- Como los diseñadores de Java desean que sea lo más portable posible, utilizan estándares reconocidos internacionalmente para los formatos de caracteres (Unicode) y números de punto flotante (IEEE 754).

Sección 4.14 (Opcional) Ejemplo práctico de GUI y gráficos: creación de dibujos simples

- El sistema de coordenadas de Java proporciona un esquema para identificar cada punto en la pantalla. De manera predeterminada, la esquina superior izquierda de un componente de la GUI tiene las coordenadas (0, 0).

- Un par de coordenadas se compone de una coordenada *x* (la coordenada horizontal) y una coordenada *y* (la coordenada vertical). La coordenada *x* es la ubicación horizontal que avanza de izquierda a derecha. La coordenada *y* es la ubicación vertical que avanza de arriba hacia abajo.
- El eje *x* describe a todas las coordenadas horizontales, y el eje *y* a todas las coordenadas verticales.
- Las unidades de las coordenadas se miden en píxeles. Un píxel es la unidad más pequeña de resolución de una pantalla.
- La clase `Graphics` (del paquete `java.awt`) proporciona varios métodos para dibujar texto y figuras en la pantalla.
- La clase `JPanel` (del paquete `javax.swing`) proporciona un área en la que un programa puede hacer dibujos.
- La palabra clave `extends` indica que una clase hereda de otra clase. La nueva clase empieza con los miembros existentes (datos y métodos) de la clase existente.
- La clase a partir de la cual la nueva clase hereda se conoce como la superclase, y la nueva clase se llama subclase.
- Todo objeto `JPanel` tiene un método `paintComponent`, que el sistema llama automáticamente cada vez que necesita mostrar el objeto `JPanel`: cuando se muestra un `JPanel` por primera vez en la pantalla, cuando una ventana en la pantalla lo cubre y luego lo descubre, y cuando se cambia el tamaño de la ventana en la que aparece este objeto.
- El método `paintComponent` requiere un argumento (un objeto `Graphics`), que el sistema proporciona por usted cuando llama a `paintComponent`.
- La primera instrucción en cualquier método `paintComponent` que usted vaya a crear debe ser siempre

```
super.paintComponent( g );
```

  Esto asegura que el panel se despliegue de manera apropiada en la pantalla, antes de empezar a dibujar en él.
- Los métodos `getWidth` y `getHeight` de `JPanel` devuelven la anchura y la altura de un objeto `JPanel`, respectivamente.
- El método `drawLine` de `Graphics` dibuja una línea entre dos puntos representados por sus cuatro argumentos. Los primeros dos argumentos son las coordenadas *x* y *y* para un punto final de la línea, y los últimos dos argumentos son las coordenadas para el otro punto final de la línea.
- Para mostrar un objeto `JPanel` en la pantalla, debe colocarlo en una ventana. Para crear una ventana, utilice un objeto de la clase `JFrame`, del paquete `javax.swing`.
- El método `setDefaultCloseOperation` de `JFrame` con el argumento `JFrame.EXIT_ON_CLOSE` indica que la aplicación debe terminar cuando el usuario cierre la ventana.
- El método `add` de `JFrame` adjunta un componente de la GUI a un objeto `JFrame`.
- El método `setSize` de `JFrame` establece la anchura y la altura del objeto `JFrame`.

Terminología

`--`, operador
`?:`, operador
`++`, operador
`+=`, operador
acción
actividad (en UML)
`add`, método de la clase `JFrame` (GUI)
algoritmo
anidamiento de estructuras de control
apilamiento de estructuras de control
bloque
`boolean`, expresión
`boolean`, tipo primitivo
ciclo
ciclo infinito
cima
círculo relleno (en UML)
circunferencia (en UML)
condición de continuación de ciclo
condición de guardia (en UML)
contador
contador de ciclo
control del programa
conversión explícita
conversión implícita
coordenada horizontal (GUI)
coordenada vertical (GUI)
coordenada *x*
coordenada *y*
cuerpo de un ciclo
decisión
diagrama de actividad (en UML)
división entera
`drawLine`, método de la clase `Graphics` (GUI)
eje *x*
eje *y*
ejecución secuencial
error de sintaxis
error fatal
error lógico
error lógico fatal
error lógico no fatal
estado de acción (en UML)
estado final (en UML)

estado inicial (en UML)
estructura de repetición
estructura de secuencia
estructura de selección
expresión condicional
expresión de acción (en UML)
`extends`
`false`
flecha de transición (en UML)
flujo de trabajo
`getHeight`, método de la clase `JPanel` (GUI)
`getWidth`, método de la clase `JPanel` (GUI)
`goto`, instrucción
`Graphics`, clase (GUI)
heredar de una clase existente
`if`, instrucción de selección simple
`if...else`, instrucción de selección doble
inicialización
instrucción de ciclo
instrucción de control
instrucción de repetición
instrucción de selección
instrucción de selección doble
instrucción de selección múltiple
instrucción de selección simple
instrucciones de control anidadas
instrucciones de control apiladas
instrucciones de control de una sola entrada/una sola salida
instrucciones `if...else` anidadas
iteración
`JComponent`, clase (GUI)
`JFrame`, clase (GUI)
`JPanel`, clase (GUI)
lenguaje fuertemente tipificado
línea punteada (en UML)
modelo de programación acción/decisión
nota (en UML)
operador condicional (`?:`)
operador de asignación compuesto
operador de asignación compuesto de suma (`+=`)
operador de conversión de tipos, (`double`)
operador de conversión de tipos, (*tipo*)
operador de conversión de tipos unario
operador de decremento (`--`)
operador de incremento (`++`)
operador de multiplicación
operador de postdecremento
operador de postincremento
operador de predecremento
operador de preincremento
operador ternario
operadores de asignación compuestos aritméticos: `+=`, `-=`, `*=`, `/=` y `%=`
orden en el que deben ejecutarse las acciones
`paintComponent`, método de la clase `JComponent` (GUI)
pequeño círculo (en UML)
píxel (GUI)
postdecrementar una variable
postincrementar una variable
predecrementar una variable
preincrementar una variable
primer refinamiento
problema del `else` suelto
procedimiento para resolver un problema
programación estructurada
promoción
refinamiento de arriba a abajo, paso a paso
repetición
repetición controlada por centinela
repetición controlada por un contador
repetición definida
repetición indefinida
rombo (en UML)
segundo refinamiento
`setDefaultCloseOperation`, método de la clase `JFrame` (GUI)
`setSize`, método de la clase `JFrame` (GUI)
seudocódigo
símbolo de decisión (en UML)
símbolo de estado de acción (en UML)
símbolo de fusión (en UML)
sistema de coordenadas (GUI)
tipos primitivos
total
transferencia de control
transición (en UML)
`true`
truncar la parte fraccionaria de un cálculo
valor centinela
valor de bandera
valor de prueba
valor de señal
variable de control
`while`, instrucción de repetición

Ejercicios de autoevaluación

4.1 Complete los siguientes enunciados:

a) Todos los programas pueden escribirse en términos de tres tipos de estructuras de control: ___________, ___________ y ___________.

b) La instrucción ___________ se utiliza para ejecutar una acción cuando una condición es verdadera, y otra acción cuando esa condición es falsa.

c) Al proceso de repetir un conjunto de instrucciones un número específico de veces se le llama repetición ___________.
d) Cuando no se sabe de antemano cuántas veces se repetirá un conjunto de instrucciones, se puede usar un valor ___________ para terminar la repetición.
e) La estructura ___________ está integrada en Java; de manera predeterminada, las instrucciones se ejecutan en el orden en el que aparecen.
f) Todas las variables de instancia de los tipos `char`, `byte`, `short`, `int`, `long`, `float` y `double` reciben el valor ___________ de manera predeterminada.
g) Java es un lenguaje ___________; requiere que todas las variables tengan un tipo.
h) Si el operador de incremento se ___________ de una variable, ésta se incrementa en 1 primero, y después su nuevo valor se utiliza en la expresión.

4.2 Conteste con *verdadero* o *falso* a cada una de las siguientes proposiciones; en caso de ser *falso*, explique por qué.

a) Un algoritmo es un procedimiento para resolver un problema en términos de las acciones a ejecutar y el orden en el que se ejecutan.
b) Un conjunto de instrucciones contenidas dentro de un par de paréntesis se denomina bloque.
c) Una instrucción de selección especifica que una acción se repetirá, mientras cierta condición siga siendo verdadera.
d) Una instrucción de control anidada aparece en el cuerpo de otra instrucción de control.
e) Java cuenta con los operadores de asignación compuestos aritméticos `+=`, `-=`, `*=`, `/=` y `%=` para abreviar las expresiones de asignación.
f) Los tipos primitivos (`boolean`, `char`, `byte`, `short`, `int`, `long`, `float` y `double`) son portables sólo en las plataformas Windows.
g) Al proceso de especificar el orden en el que se ejecutan las instrucciones (acciones) en un programa se denomina control del programa.
h) El operador de conversión de tipos unario (`double`) crea una copia entera temporal de su operando.
i) Las variables de instancia de tipo `boolean` reciben el valor `true` de manera predeterminada.
j) El seudocódigo ayuda a un programador a idear un programa, antes de tratar de escribirlo en un lenguaje de programación.

4.3 Escriba cuatro instrucciones distintas en Java, en donde cada una sume 1 a la variable entera `x`.

4.4 Escriba instrucciones en Java para realizar cada una de las siguientes tareas:

a) Asignar la suma de `x` e `y` a `z`, e incrementar `x` en 1 después del cálculo. Use sólo una instrucción.
b) Evaluar si la variable `cuenta` es mayor que 10. De ser así, imprimir `"Cuenta es mayor que 10"`.
c) Decrementar la variable `x` en 1, luego restarla a la variable `total`. Use sólo una instrucción.
d) Calcular el residuo después de dividir `q` entre `divisor`, y asignar el resultado a `q`. Escriba esta instrucción de dos maneras distintas.

4.5 Escriba una instrucción en Java para realizar cada una de las siguientes tareas:

a) Declarar las variables `suma` y `x` como de tipo `int`.
b) Asignar 1 a la variable `x`.
c) Asignar 0 a la variable `suma`.
d) Sumar la variable `x` a `suma` y asignar el resultado a la variable `suma`.
e) Imprimir la cadena `"La suma es: "`, seguida del valor de la variable `suma`.

4.6 Combine las instrucciones que escribió en el ejercicio 4.5 para formar una aplicación en Java que calcule e imprima la suma de los enteros del 1 al 10. Use una instrucción `while` para iterar a través de las instrucciones de cálculo e incremento. El ciclo debe terminar cuando el valor de `x` se vuelva 11.

4.7 Determine el valor de las variables en la siguiente instrucción, después de realizar el cálculo. Suponga que, cuando se empieza a ejecutar la instrucción, todas las variables son de tipo `int` y tienen el valor 5.

```
producto *= x++;
```

4.8 Identifique y corrija los errores en cada uno de los siguientes fragmentos de código:

a)
```
while ( c <= 5 )
{
   producto *= c;
   ++c;
```
b)
```
if ( genero == 1 )
   System.out.println( "Mujer" );
else;
   System.out.println( "Hombre" );
```

4.9 ¿Qué está mal en la siguiente instrucción `while`?

```
while ( z >= 0 )
   suma += z;
```

Respuestas a los ejercicios de autoevaluación

4.1 a) secuencia, selección, repetición. b) `if...else`. c) controlada por contador (o definida). d) centinela, de señal, de prueba o de bandera. e) secuencia. f) `0` (cero). g) fuertemente tipificado. h) coloca antes.

4.2 a) Verdadero. b) Falso. Un conjunto de instrucciones contenidas dentro de un par de llaves (`{` y `}`) se denomina bloque. c) Falso. Una instrucción de repetición especifica que una acción se repetirá mientras que cierta condición siga siendo verdadera. d) Verdadero. e) Verdadero. f) Falso. Los tipos primitivos (`boolean`, `char`, `byte`, `short`, `int`, `long`, `float` y `double`) son portables a través de todas las plataformas de computadora que soportan Java. g) Verdadero. h) Falso. El operador de conversión de tipos unario (`double`) crea una copia temporal de punto flotante de su operando. i) Falso. Las variables de instancia de tipo `boolean` reciben el valor `false` de manera predeterminada. j) Verdadero.

4.3
```
x = x + 1;
x += 1;
++x;
x++;
```

4.4 a) `z = x++ + y;`

b)
```
if ( cuenta > 10 )
   System.out.println( "Cuenta es mayor que 10" );
```
c) `total -= --x;`

d)
```
q %= divisor;
q = q % divisor;
```

4.5 a) `int suma, x;`

b) `x = 1;`

c) `suma = 0;`

d) `suma += x;` o `suma = suma + x;`

e) `System.out.printf( "La suma es: %d\n", suma );`

4.6 El programa se muestra a continuación:

```
// Calcula la suma de los enteros del 1 al 10
public class Calcular
{
   public static void main( String args[] )
   {
      int suma;
      int x;

      x = 1;   // inicializa x en 1 para contar
      suma = 0; // inicializa suma en 0 para el total

      while ( x <= 10 ) // mientras que x sea menor o igual que 10
      {
         suma += x; // suma x a suma
         ++x; // incrementa x
```

```
        } // fin de while

        System.out.printf( "La suma es: %d\n", suma );
    } // fin de main

} // fin de la clase Calcular
```

```
La suma es: 55
```

4.7 `producto = 25, x = 6`

4.8 a) Error: falta la llave derecha de cierre del cuerpo de la instrucción `while`.
Corrección: Agregar una llave derecha de cierre después de la instrucción `++c;`.
b) Error: El punto y coma después de `else` produce un error lógico. La segunda instrucción de salida siempre se ejecutará.
Corrección: Quitar el punto y coma después de `else`.

4.9 El valor de la variable `z` nunca se cambia en la instrucción `while`. Por lo tanto, si la condición de continuación de ciclo `( z >= 0 )` es verdadera, se crea un ciclo infinito. Para evitar que ocurra un ciclo infinito, `z` debe decrementarse de manera que eventualmente se vuelva menor que 0.

Ejercicios

4.10 Compare y contraste la instrucción `if` de selección simple y la instrucción de repetición `while`. ¿Cuál es la similitud en las dos instrucciones? ¿Cuál es su diferencia?

4.11 Explique lo que ocurre cuando un programa en Java trata de dividir un entero entre otro. ¿Qué ocurre con la parte fraccionaria del cálculo? ¿Cómo puede un programador evitar ese resultado?

4.12 Describa las dos formas en las que pueden combinarse las instrucciones de control.

4.13 ¿Qué tipo de repetición sería apropiada para calcular la suma de los primeros 100 enteros positivos? ¿Qué tipo de repetición sería apropiada para calcular la suma de un número arbitrario de enteros positivos? Describa brevemente cómo podría realizarse cada una de estas tareas.

4.14 ¿Cuál es la diferencia entre preincrementar y postincrementar una variable?

4.15 Identifique y corrija los errores en cada uno de los siguientes fragmentos de código. [*Nota:* puede haber más de un error en cada fragmento de código].

a)
```
if ( edad >= 65 );
   System.out.println( "Edad es mayor o igual que 65" );
else
   System.out.println( "Edad es menor que 65 )";
```
b)
```
int x = 1, total;
while ( x <= 10 )
{
   total += x;
   ++x;
}
```
c)
```
while ( x <= 100 )
   total += x;
   ++x;
```
d)
```
while ( y > 0 )
{
   System.out.println( y );
   ++y;
```

4.16 ¿Qué es lo que imprime el siguiente programa?

```
public class Misterio
{
    public static void main( String args[] )
    {
       int y;
       int x = 1;
    int total = 0;

    while ( x <= 10 )
    {
       y = x * x;
       System.out.println( y );
       total += y;
       ++x;
    } // fin de while

    System.out.printf( "El total es %d\n", total );
    } // fin de main

} // fin de la clase Misterio
```

Para los ejercicios 4.17 a 4.20, realice cada uno de los siguientes pasos:

a) Lea el enunciado del problema.
b) Formule el algoritmo utilizando seudocódigo y el proceso de refinamiento de arriba a abajo, paso a paso.
c) Escriba un programa en Java.
d) Pruebe, depure y ejecute el programa en Java.
e) Procese tres conjuntos completos de datos.

4.17 Los conductores se preocupan acerca del kilometraje de sus automóviles. Un conductor ha llevado el registro de varios reabastecimientos de gasolina, registrando los kilómetros conducidos y los litros usados en cada reabastecimiento. Desarrolle una aplicación en Java que reciba como entrada los kilómetros conducidos y los litros usados (ambos como enteros) por cada reabastecimiento. El programa debe calcular y mostrar los kilómetros por litro obtenidos en cada reabastecimiento, y debe imprimir el total de kilómetros por litro obtenidos en todos los reabastecimientos hasta este punto. Todos los cálculos del promedio deben producir resultados en números de punto flotante. Use la clase `Scanner` y la repetición controlada por centinela para obtener los datos del usuario.

4.18 Desarrolle una aplicación en Java que determine si alguno de los clientes de una tienda de departamentos se ha excedido del límite de crédito en una cuenta. Para cada cliente se tienen los siguientes datos:

a) El número de cuenta.
b) El saldo al inicio del mes.
c) El total de todos los artículos cargados por el cliente en el mes.
d) El total de todos los créditos aplicados a la cuenta del cliente en el mes.
e) El límite de crédito permitido.

El programa debe recibir como entrada cada uno de estos datos en forma de números enteros, debe calcular el nuevo saldo (= *saldo inicial* + *cargos* – *créditos*), mostrar el nuevo balance y determinar si éste excede el límite de crédito del cliente. Para los clientes cuyo límite de crédito sea excedido, el programa debe mostrar el mensaje `"Se excedió el límite de su crédito"`.

4.19 Una empresa grande paga a sus vendedores mediante comisiones. Los vendedores reciben $200 por semana, más el 9% de sus ventas brutas durante esa semana. Por ejemplo, un vendedor que vende $5000 de mercancía en una semana, recibe $200 más el 9% de 5000, o un total de $650. Usted acaba de recibir una lista de los artículos vendidos por cada vendedor. Los valores de estos artículos son los siguientes:

Artículo	Valor
1	239.99
2	129.75
3	99.95
4	350.89

Desarrolle una aplicación en Java que reciba como entrada los artículos vendidos por un vendedor durante la última semana, y que calcule y muestre los ingresos de ese vendedor. No hay límite en cuanto al número de artículos que un vendedor puede vender.

4.20 Desarrolle una aplicación en Java que determine el sueldo bruto para cada uno de tres empleados. La empresa paga la cuota normal en las primeras 40 horas de trabajo de cada empleado, y paga cuota y media en todas las horas trabajadas que excedan de 40. Usted recibe una lista de los empleados de la empresa, el número de horas que trabajó cada empleado la semana pasada y la tarifa por horas de cada empleado. Su programa debe recibir como entrada esta información para cada empleado, debe determinar y mostrar el sueldo bruto de cada empleado. Utilice la clase `Scanner` para introducir los datos.

4.21 El proceso de encontrar el valor más grande (es decir, el máximo de un grupo de valores) se utiliza frecuentemente en aplicaciones de computadora. Por ejemplo, un programa para determinar el ganador de un concurso de ventas recibe como entrada el número de unidades vendidas por cada vendedor. El vendedor que haya vendido más unidades es el que gana el concurso. Escriba un programa en seudocódigo y después una aplicación en Java que reciba como entrada una serie de 10 números enteros, y que determine e imprima el mayor de los números. Su programa debe utilizar cuando menos las siguientes tres variables:

a) `contador`: un contador para contar hasta 10 (es decir, para llevar el registro de cuántos números se han introducido, y para determinar cuando se hayan procesado los 10 números).
b) `numero`: el entero más reciente introducido por el usuario.
c) `mayor`: el número más grande encontrado hasta ahora.

4.22 Escriba una aplicación en Java que utilice ciclos para imprimir la siguiente tabla de valores:

```
N       10*N    100*N    1000*N

1       10      100      1000
2       20      200      2000
3       30      300      3000
4       40      400      4000
5       50      500      5000
```

4.23 Utilizando una metodología similar a la del ejercicio 4.21, encuentre los *dos* valores más grandes de los 10 que se introdujeron. [*Nota:* puede introducir cada número sólo una vez].

4.24 Modifique el programa de la figura 4.12 para validar sus entradas. Para cualquier entrada, si el valor introducido es distinto de 1 o 2, debe seguir iterando hasta que el usuario introduzca un valor correcto.

4.25 ¿Qué es lo que imprime el siguiente programa?

```java
public class Misterio2
{
   public static void main( String args[] )
   {
     int cuenta = 1;

     while ( cuenta <= 10 )
     {
       System.out.println( cuenta % 2 == 1 ? "****" : "++++++++" );
       ++cuenta;
     } // fin de while
   } // fin de main

} // fin de la clase Misterio2
```

4.26 ¿Qué es lo que imprime el siguiente programa?

```java
public class Misterio3
{
```

```
public static void main( String args[] )
{
  int fila = 10;
  int columna;

  while ( fila >= 1 )
  {
    columna = 1;

    while ( columna <= 10 )
    {
      System.out.print( fila % 2 == 1 ? "<" : ">" );
      ++columna;
    } // fin de while

      --fila;
      System.out.println();
    } // fin de while
  } // fin de main

} // fin de la clase Misterio3
```

4.27 (*Problema del `else` suelto*) Determine la salida de cada uno de los siguientes conjuntos de código, cuando `x` es `9` y `y` es `11`, y cuando `x` es `11` y `y` es `9`. Observe que el compilador ignora la sangría en un programa en Java. Además, el compilador de Java siempre asocia un `else` con el `if` que le precede inmediatamente, a menos que se le indique de otra forma mediante la colocación de llaves (`{}`). A primera vista, el programador tal vez no esté seguro de cuál `if` corresponde a cuál `else`; esta situación se conoce como el "problema del `else` suelto". Hemos eliminado la sangría del siguiente código para hacer el problema más retador. [*Sugerencia:* aplique las convenciones de sangría que ha aprendido].

a)
```
if ( x < 10 )
if ( y > 10 )
System.out.println( "*****" );
else
System.out.println( "#####" );
System.out.println( "$$$$$" );
```

b)
```
if ( x < 10 )
{
if ( y > 10 )
System.out.println( "*****" );
}
else
{
System.out.println( "#####" );
System.out.println( "$$$$$" );
}
```

4.28 (*Otro problema de `else` suelto*) Modifique el código dado para producir la salida que se muestra en cada parte del problema. Utilice las técnicas de sangría apropiadas. No haga modificaciones en el código, sólo inserte llaves o modifique la sangría del código. El compilador ignora la sangría en un programa en Java. Hemos eliminado la sangría en el código dado, para hacer el problema más retador. [*Nota:* es posible que no se requieran modificaciones en algunas de las partes].

```
if ( y == 8 )
if ( x == 5 )
System.out.println( "@@@@@" );
else
System.out.println( "#####" );
System.out.println( "$$$$$" );
System.out.println( "&&&&&" );
```

a) Suponiendo que `x = 5` y `y = 8`, se produce la siguiente salida:

```
@@@@@
$$$$$
&&&&&
```

b) Suponiendo que `x = 5` y `y = 8`, se produce la siguiente salida:

```
@@@@@
```

c) Suponiendo que `x = 5` y `y = 8`, se produce la siguiente salida:

```
@@@@@
&&&&&
```

d) Suponiendo que `x = 5` y `y = 7`, se produce la siguiente salida. [*Nota:* las tres últimas instrucciones de salida después del `else` forman parte de un bloque.]

```
#####
$$$$$
&&&&&
```

4.29 Escriba una aplicación que pida al usuario que introduzca el tamaño del lado de un cuadrado y que muestre un cuadrado hueco de ese tamaño, compuesto de asteriscos. Su programa debe funcionar con cuadrados que tengan lados de todas las longitudes entre 1 y 20.

4.30 (*Palíndromos*) Un palíndromo es una secuencia de caracteres que se lee igual al derecho y al revés. Por ejemplo, cada uno de los siguientes enteros de cinco dígitos es un palíndromo: 12321, 55555, 45554 y 11611. Escriba una aplicación que lea un entero de cinco dígitos y determine si es un palíndromo. Si el número no es de cinco dígitos, el programa debe mostrar un mensaje de error y permitir al usuario que introduzca un nuevo valor.

4.31 Escriba una aplicación que reciba como entrada un entero que contenga sólo 0s y 1s (es decir, un entero binario), y que imprima su equivalente decimal. [*Sugerencia:* use los operadores residuo y división para elegir los dígitos del número binario uno a la vez, de derecha a izquierda. En el sistema numérico decimal, el dígito más a la derecha tiene un valor posicional de 1 y el siguiente dígito a la izquierda tiene un valor posicional de 10, después 100, después 1000, etcétera. El número decimal 234 puede interpretarse como 4 * 1 + 3 * 10 + 2 * 100. En el sistema numérico binario, el dígito más a la derecha tiene un valor posicional de 1, el siguiente dígito a la izquierda tiene un valor posicional de 2, luego 4, luego 8, etcétera. El equivalente decimal del número binario 1101 es 1 * 1 + 0 * 2 + 1 * 4 + 1 * 8, o 1 + 0 + 4 + 8, o 13].

4.32 Escriba una aplicación que utilice sólo las instrucciones de salida

```
System.out.print( "* " );
System.out.print( " " );
System.out.println();
```

para mostrar el patrón de tablero de damas que se muestra a continuación. Observe que una llamada al método `System.out.println` sin argumentos hace que el programa imprima un solo carácter de nueva línea. [*Sugerencia:* se requieren estructuras de repetición].

```
* * * * * * * *
 * * * * * * * *
* * * * * * * *
 * * * * * * * *
* * * * * * * *
 * * * * * * * *
* * * * * * * *
 * * * * * * * *
```

4.33 Escriba una aplicación que muestre en la ventana de comandos los múltiplos del entero 2 (es decir, 2, 4, 8, 16, 32, 64, etcétera). Su ciclo no debe terminar (es decir, debe crear un ciclo infinito). ¿Qué ocurre cuando ejecuta este programa?

4.34 ¿Qué está mal en la siguiente instrucción? Proporcione la instrucción correcta para sumar uno a la suma de `x` e `y`.

```
System.out.println( ++(x + y) );
```

4.35 Escriba una aplicación que lea tres valores distintos de cero introducidos por el usuario, y que determine e imprima si podrían representar los lados de un triángulo.

4.36 Escriba una aplicación que lea tres enteros distintos de cero, determine e imprima si estos enteros podrían representar los lados de un triángulo rectángulo.

4.37 Una compañía desea transmitir datos a través del teléfono, pero le preocupa que sus teléfonos puedan estar intervenidos. Le ha pedido a usted que escriba un programa que cifre sus datos, de manera que éstos puedan transmitirse con más seguridad. Todos los datos se transmiten como enteros de cuatro dígitos. Su aplicación debe leer un entero de cuatro dígitos introducido por el usuario y cifrarlo de la siguiente manera: reemplace cada dígito con el resultado de sumar 7 al dígito y obtener el residuo después de dividir el nuevo valor entre 10. Luego intercambie el primer dígito con el tercero, e intercambie el segundo dígito con el cuarto. Después imprima el entero cifrado. Escriba una aplicación separada que reciba como entrada un entero de cuatro dígitos cifrado, y que lo descifre para formar el número original.

4.38 El factorial de un entero n no negativo se escribe como $n!$ y se define de la siguiente manera:

$$n! = n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 1 \quad \text{(para valores de } n \text{ mayores o iguales a 1)}$$

y

$$n! = 1 \quad \text{(para } n = 0\text{)}$$

Por ejemplo, $5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$, que es 120.

a) Escriba una aplicación que lea un entero no negativo, y calcule e imprima su factorial.

b) Escriba una aplicación que estime el valor de la constante matemática e, utilizando la fórmula

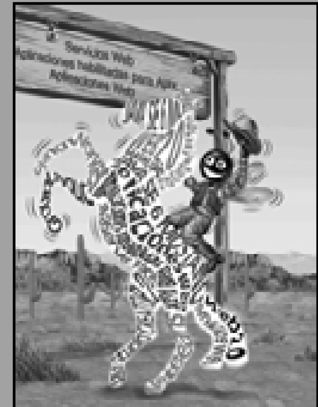
$$e = 1 + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \ldots$$

c) Escriba una aplicación que calcule el valor de e^x, utilizando la fórmula

$$e^x = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots$$

5

Instrucciones de control: parte 2



No todo lo que puede contarse cuenta, y no todo lo que cuenta puede contarse.

—Albert Einstein

¿Quién puede controlar su destino?

—William Shakespeare

La llave usada siempre es brillante.

—Benjamin Franklin

La inteligencia... es la facultad de hacer objetos artificiales, especialmente herramientas para hacer herramientas.

—Henri Bergson

Cada ventaja en el pasado se juzga a la luz de la cuestión final.

—Demóstenes

OBJETIVOS

En este capítulo aprenderá a:

- Conocer los fundamentos acerca de la repetición controlada por un contador.
- Utilizar las instrucciones de repetición `for` y `do...while` para ejecutar instrucciones de manera repetitiva en un programa.
- Comprender la selección múltiple utilizando la instrucción de selección `switch`.
- Utilizar las instrucciones de control de programa `break` y `continue` para alterar el flujo de control.
- Utilizar los operadores lógicos para formar expresiones condicionales complejas en instrucciones de control.

Plan general

5.1 Introducción

El capítulo 4 nos introdujo a los tipos de bloques de construcción disponibles para solucionar problemas. Utilizamos dichos bloques de construcción para emplear las técnicas, ya comprobadas, de la construcción de programas. En este capítulo continuaremos nuestra presentación de la teoría y los principios de la programación estructurada, presentando el resto de las instrucciones de control en Java. Las instrucciones de control que estudiaremos aquí y las que vimos en el capítulo 4 son útiles para crear y manipular objetos.

En este capítulo demostraremos las instrucciones `for`, `do...while` y `switch` de Java. A través de una serie de ejemplos cortos en los que utilizaremos las instrucciones `while` y `for`, exploraremos los fundamentos acerca de la repetición controlada por contador. Dedicaremos una parte de este capítulo (y del capítulo 7) a expandir la clase `LibroCalificaciones` que presentamos en los capítulos 3 y 4. En especial, crearemos una versión de la clase `LibroCalificaciones` que utiliza una instrucción `switch` para contar el número de calificaciones equivalentes de A, B, C, D y F, en un conjunto de calificaciones numéricas introducidas por el usuario. Presentaremos las instrucciones de control de programa `break` y `continue`. Hablaremos sobre los operadores lógicos de Java, que nos permiten utilizar expresiones condicionales más complejas en las instrucciones de control. Por último, veremos un resumen de las instrucciones de control de Java y las técnicas ya probadas de solución de problemas que presentamos en éste y en el capítulo 4.

5.2 Fundamentos de la repetición controlada por contador

Esta sección utiliza la instrucción de repetición `while`, presentada en el capítulo 4, para formalizar los elementos requeridos y llevar a cabo la repetición controlada por contador. Este tipo de repetición requiere:

1. Una **variable de control** (o contador de ciclo).
2. El **valor inicial** de la variable de control.
3. El **incremento** (o **decremento**) con el que se modifica la variable de control cada vez que pasa por el ciclo (lo que también se conoce como **cada iteración del ciclo**).
4. La **condición de continuación de ciclo**, que determina si el ciclo debe continuar o no.

Para ver estos elementos de la repetición controlada por contador, considere la aplicación de la figura 5.1, que utiliza un ciclo para mostrar los números del 1 al 10. Observe que esta figura sólo contiene un método, `main`, que realiza todo el trabajo de la clase. Para la mayoría de las aplicaciones, en los capítulos 3 y 4 hemos preferido el uso de dos archivos separados: uno que declara una clase reutilizable (por ejemplo, `Cuenta`) y otro que instancia uno o más objetos de esa clase (por ejemplo, `PruebaCuenta`) y demuestra su funcionalidad. Sin embargo, en algunas ocasiones es más apropiado crear sólo una clase, cuyo método `main` ilustra en forma concisa un concepto básico. A lo largo de este capítulo, utilizaremos varios ejemplos de una clase, como el de la figura 5.1, para demostrar la mecánica de las instrucciones de control de Java.

En la figura 5.1, los elementos de la repetición controlada por contador se definen en las líneas 8, 10 y 13. La línea 8 declara la variable de control (`contador`) como un `int`, reserva espacio para esta variable en memoria y establece su valor inicial en `1`. La variable `contador` también podría haberse declarado e inicializado con la siguientes instrucciones de declaración y asignación de variables locales:

```
int contador; // declara contador
contador = 1; // inicializa contador en 1
```

La línea 12 muestra el valor de la variable de control `contador` durante cada iteración del ciclo. La línea 13 incrementa la variable de control en 1, para cada iteración del ciclo. La condición de continuación de ciclo en la instrucción `while` (línea 10) evalúa si el valor de la variable de control es menor o igual que `10` (el valor final para el que la condición es `true`). Observe que el programa ejecuta el cuerpo de este `while` aun y cuando la variable de control sea `10`. El ciclo termina cuando la variable de control es mayor a `10` (es decir, cuando `contador` se convierte en `11`).



Error común de programación 5.1

Debido a que los valores de punto flotante pueden ser aproximados, controlar los ciclos con variables de punto flotante puede producir valores imprecisos del contador y pruebas de terminación imprecisas.



Tip para prevenir errores 5.1

Controle los ciclos de contador con enteros.



Buena práctica de programación 5.1

Coloque líneas en blanco por encima y debajo de las instrucciones de control de repetición y selección, y aplique sangría a los cuerpos de las instrucciones para mejorar la legibilidad.

El programa de la figura 5.1 puede hacerse más conciso si inicializamos `contador` en `0` en la línea 8, y preincrementamos `contador` en la condición `while` de la siguiente forma:

```
while ( ++contador <= 10 ) // condición de continuación de ciclo
   System.out.printf( "%d ", contador );
```

```
1  // Fig. 5.1: ContadorWhile.java
2  // Repetición controlada con contador, con la instrucción de repetición while.
3
4  public class ContadorWhile
5  {
6     public static void main( String args[] )
7     {
8        int contador = 1; // declara e inicializa la variable de control
9
10       while ( contador <= 10 ) // condición de continuación de ciclo
11       {
12          System.out.printf( "%d  ", contador );
13          ++contador; // incrementa la variable de control en 1
14       } // fin de while
15
16       System.out.println(); // imprime una nueva línea
17    } // fin de main
18 } // fin de la clase ContadorWhile
```

```
1  2  3  4  5  6  7  8  9  10
```

Figura 5.1 | Repetición controlada con contador, con la instrucción de repetición `while`.

Este código ahorra una instrucción (y elimina la necesidad de usar llaves alrededor del cuerpo del ciclo), ya que la condición de `while` realiza el incremento antes de evaluar la condición. (En la sección 4.12 vimos que la precedencia de ++ es mayor que la de <=). La codificación en esta forma tan condensada requiere de práctica y puede hacer que el código sea más difícil de leer, depurar, modificar y mantener, así que en general, es mejor evitarla.



Observación de ingeniería de software 5.1

"Mantener las cosas simples" es un buen consejo para la mayoría del código que usted escriba.

5.3 Instrucción de repetición for

La sección 5.2 presentó los aspectos esenciales de la repetición controlada por contador. La instrucción `while` puede utilizarse para implementar cualquier ciclo controlado por un contador. Java también cuenta con la instrucción de repetición `for`, que especifica los detalles de la repetición controlada por contador en una sola línea de código. La figura 5.2 reimplementa la aplicación de la figura 5.1, usando la instrucción `for`.

El método `main` de la aplicación opera de la siguiente manera: cuando la instrucción `for` (líneas 10 y 11) comienza a ejecutarse, la variable de control `contador` se declara e inicializa en `1` (en la sección 5.2 vimos que los primeros dos elementos de la repetición controlada por un contador son la variable de control y su valor inicial). A continuación, el programa verifica la condición de continuación de ciclo, `contador <= 10`, la cual se encuentra entre los dos signos de punto y coma requeridos. Como el valor inicial de `contador` es `1`, al principio la condición es verdadera. Por lo tanto, la instrucción del cuerpo (línea 11) muestra el valor de la variable de control `contador`, que es `1`. Después de ejecutar el cuerpo del ciclo, el programa incrementa a `contador` en la expresión `contador++`, la cual aparece a la derecha del segundo signo de punto y coma. Después, la prueba de continuación de ciclo se ejecuta de nuevo para determinar si el programa debe continuar con la siguiente iteración del ciclo. En este punto, el valor de la variable de control es `2`, por lo que la condición sigue siendo verdadera (el valor final no se excede); así, el programa ejecuta la instrucción del cuerpo otra vez (es decir, la siguiente iteración del ciclo). Este proceso continúa hasta que se muestran en pantalla los números del 1 al 10 y el valor de `contador` se vuelve `11`, con lo cual falla la prueba de continuación de ciclo y termina la repetición (después de 10 repeticiones del cuerpo del ciclo en la línea 11). Después, el programa ejecuta la primera instrucción después del `for`; en este caso, la línea 13.

Observe que la figura 5.2 utiliza (en la línea 10) la condición de continuación de ciclo `contador <= 10`. Si usted especificara por error `contador < 10` como la condición, el ciclo sólo iteraría nueve veces. A este error lógico común se le conoce como **error por desplazamiento en 1**.

```
1  // Fig. 5.2: ContadorFor.java
2  // Repetición controlada con contador, con la instrucción de repetición for.
3
4  public class ContadorFor
5  {
6     public static void main( String args[] )
7     {
8        // el encabezado de la instrucción for incluye la inicialización,
9        // la condición de continuación de ciclo y el incremento
10       for ( int contador = 1; contador <= 10; contador++ )
11          System.out.printf( "%d  ", contador );
12
13       System.out.println(); // imprime una nueva línea
14    } // fin de main
15 } // fin de la clase ContadorFor
```

```
1  2  3  4  5  6  7  8  9  10
```

Figura 5.2 | Repetición controlada con contador, con la instrucción de repetición `for`.



Error común de programación 5.2

Utilizar un operador relacional incorrecto o un valor final incorrecto de un contador de ciclo en la condición de continuación de ciclo de una instrucción de repetición puede producir un error por desplazamiento en 1.



Buena práctica de programación 5.2

Utilizar el valor final en la condición de una instrucción `while` *o* `for` *con el operador relacional* `<=` *nos ayuda a evitar los errores por desplazamiento en 1. Para un ciclo que imprime los valores del 1 al 10, la condición de continuación de ciclo debe ser* `contador <= 10`*, en vez de* `contador < 10` *(lo cual produce un error por desplazamiento en uno) o* `contador <11` *(que es correcto). Muchos programadores prefieren el llamado conteo con base cero, en el cual para contar 10 veces,* `contador` *se inicializaría en cero y la prueba de continuación de ciclo sería* `contador < 10`*.*

La figura 5.3 analiza con más detalle la instrucción `for` de la figura 5.2. A la primera línea del `for` (incluyendo la palabra clave `for` y todo lo que está entre paréntesis después de ésta), la línea 10 de la figura 5.2, se le llama algunas veces **encabezado de la instrucción `for`**, o simplemente **encabezado** del `for`. Observe que el encabezado del `for` "se encarga de todo": especifica cada uno de los elementos necesarios para la repetición controlada por un contador con una variable de control. Si hay más de una instrucción en el cuerpo del `for`, se requieren llaves (`{` y `}`) para definir el cuerpo del ciclo.

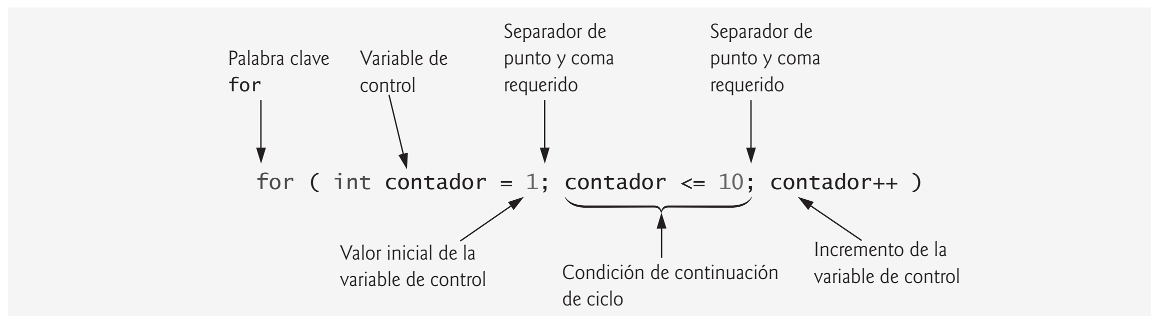


Figura 5.3 | Componentes del encabezado de la instrucción `for`.

El formato general de la instrucción `for` es

`for (` *inicialización*`;` *condiciónDeContinuaciónDeCiclo*`;` *incremento* `)`
    instrucción

en donde la expresión *inicialización* nombra a la variable de control de ciclo y proporciona su valor inicial, la *condiciónDeContinuaciónDeCiclo* es la condición que determina si el ciclo debe seguir ejecutándose, y el *incremento* modifica el valor de la variable de control (ya sea un incremento o un decremento), de manera que la condición de continuación de ciclo se vuelva falsa en un momento dado. Los dos signos de punto y coma en el encabezado del `for` son requeridos.



Error común de programación 5.3

Utilizar comas en vez de los dos signos de punto y coma requeridos en el encabezado de una instrucción `for` *es un error de sintaxis.*

En la mayoría de los casos, la instrucción `for` puede representarse con una instrucción `while` equivalente, de la siguiente manera:

inicialización`;`

`while (` *condiciónDeContinuaciónDeCiclo* `)`

```
{
    instrucción
    incremento;
}
```

En la sección 5.7 veremos un caso para el cual no es posible representar una instrucción `for` con una instrucción `while` equivalente.

Por lo general, las instrucciones `for` se utilizan para la repetición controlada por un contador, y las instrucciones `while` se utilizan para la repetición controlada por un centinela. No obstante, `while` y `for` pueden utilizarse para cualquiera de los dos tipos de repetición.

Si la expresión de *inicialización* en el encabezado del `for` declara la variable de control (es decir, si el tipo de la variable de control se especifica antes del nombre de la variable, como en la figura 5.2), la variable de control puede utilizarse sólo en esa instrucción `for`; no existirá fuera de esta instrucción. Este uso restringido del nombre de la variable de control se conoce como el **alcance** de la variable. El alcance de una variable define en dónde puede utilizarse en un programa. Por ejemplo, una variable local sólo puede utilizarse en el método que declara a esa variable, y sólo a partir del punto de declaración, hasta el final del método. En el capítulo 6, Métodos: un análisis más detallado, veremos con detalle el concepto de alcance.



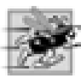
Error común de programación 5.4

Cuando se declara la variable de control de una instrucción `for` en la sección de inicialización del encabezado del `for`, si se utiliza la variable de control fuera del cuerpo del `for` se produce un error de compilación.

Las tres expresiones en un encabezado `for` son opcionales. Si se omite la *condiciónDeContinuaciónDeCiclo*, Java asume que esta condición siempre será verdadera, con lo cual se crea un ciclo infinito. Podríamos omitir la expresión de *inicialización* si el programa inicializa la variable de control antes del ciclo. Podríamos omitir la expresión de *incremento* si el programa calcula el incremento mediante instrucciones dentro del cuerpo del ciclo, o si no se necesita un incremento. La expresión de incremento en un `for` actúa como si fuera una instrucción independiente al final del cuerpo del `for`. Por lo tanto, las expresiones

```
contador = contador + 1
contador += 1
++contador
contador++
```

son expresiones de incremento equivalentes en una instrucción `for`. Muchos programadores prefieren `contador++`, ya que es concisa y además un ciclo `for` evalúa su expresión de incremento después de la ejecución de su cuerpo. Por ende, la forma de postincremento parece más natural. En este caso, la variable que se incrementa no aparece en una expresión más grande, por lo que los operadores de preincremento y postdecremento tienen en realidad el mismo efecto.



Tip de rendimiento 5.1

Hay una ligera ventaja de rendimiento al utilizar el operador de preincremento, pero si elije el operador de postincremento debido a que parece ser más natural (como en el encabezado de un `for`), los compiladores con optimización generarán código byte de Java que utilice la forma más eficiente, de todas maneras.



Buena práctica de programación 5.3

En muchos casos, los operadores de preincremento y postincremento se utilizan para sumar 1 a una variable en una instrucción por sí sola. En estos casos el efecto es idéntico, sólo que el operador de preincremento tiene una ligera ventaja de rendimiento. Dado que el compilador por lo general optimiza el código que usted escribe para ayudarlo a obtener el mejor rendimiento, puede usar cualquiera de los dos operadores (preincremento o postincremento) con el que se sienta más cómodo en estas situaciones.



Error común de programación 5.5

Colocar un punto y coma inmediatamente a la derecha del paréntesis derecho del encabezado de un `for` convierte el cuerpo de ese `for` en una instrucción vacía. Por lo general esto es un error lógico.



Tip para prevenir errores 5.2

Los ciclos infinitos ocurren cuando la condición de continuación de ciclo en una instrucción de repetición nunca se vuelve `false`. *Para evitar esta situación en un ciclo controlado por un contador, debe asegurarse que la variable de control se incremente (o decremente) durante cada iteración del ciclo. En un ciclo controlado por centinela, asegúrese que el valor centinela se introduzca en algún momento dado.*

Las porciones correspondientes a la inicialización, la condición de continuación de ciclo y el incremento de una instrucción `for` pueden contener expresiones aritméticas. Por ejemplo, suponga que `x = 2` y `y = 10`; si `x` y `y` no se modifican en el cuerpo del ciclo, entonces la instrucción

```
for ( int j = x; j <= 4 * x * y; j += y / x )
```

es equivalente a la instrucción

```
for ( int j = 2; j <= 80; j += 5 )
```

El incremento de una instrucción `for` también puede ser negativo, en cuyo caso sería un decremento y el ciclo contaría en orden descendente.

Si al principio la condición de continuación de ciclo es `false`, el programa no ejecutará el cuerpo de la instrucción `for`, sino que la ejecución continuará con la instrucción que siga inmediatamente después del `for`.

Con frecuencia, los programas muestran en pantalla el valor de la variable de control o lo utilizan en cálculos dentro del cuerpo del ciclo, pero este uso no es obligatorio. Por lo general, la variable de control se utiliza para controlar la repetición sin que se le mencione dentro del cuerpo del `for`.



Tip para prevenir errores 5.3

Aunque el valor de la variable de control puede cambiarse en el cuerpo de un ciclo `for`, *evite hacerlo, ya que esta práctica puede llevarlo a cometer errores sutiles.*

El diagrama de actividad de UML de la instrucción for es similar al de la instrucción while (figura 4.4). La figura 5.4 muestra el diagrama de actividad de la instrucción for de la figura 5.2. El diagrama hace evidente que la inicialización ocurre sólo una vez antes de evaluar la condición de continuación de ciclo por primera vez, y que el incremento ocurre cada vez que se realiza una iteración, después de que se ejecuta la instrucción del cuerpo.

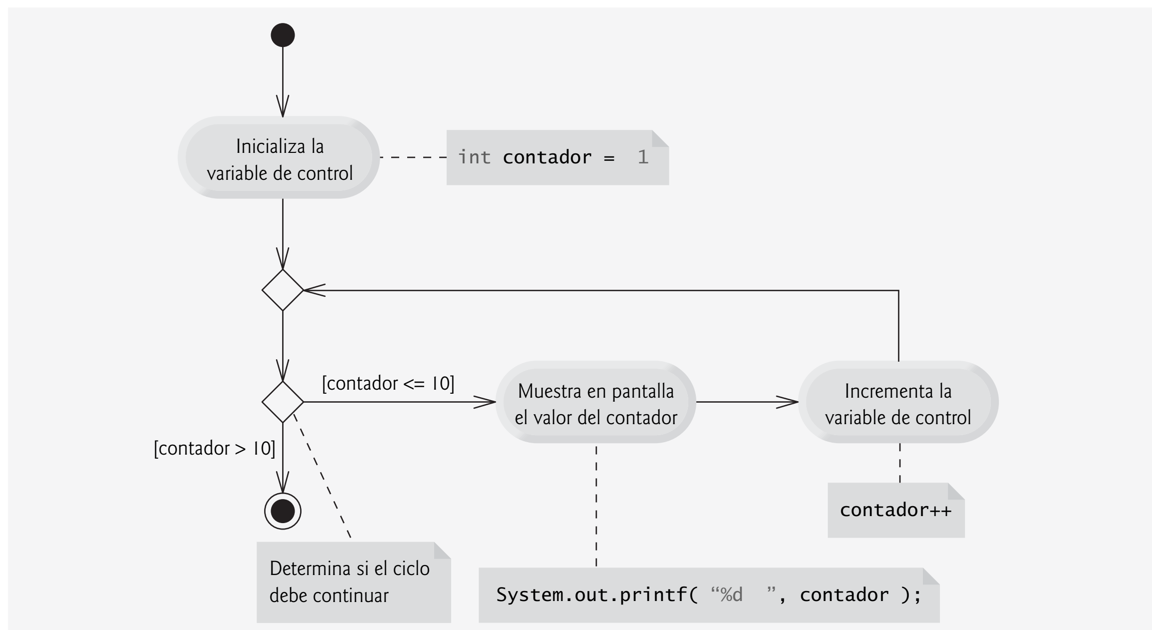


Figura 5.4 | Diagrama de actividad de UML para la instrucción `for` de la figura 5.2.

5.4 Ejemplos sobre el uso de la instrucción for

Los siguientes ejemplos muestran técnicas para modificar la variable de control en una instrucción `for`. En cada caso, escribimos el encabezado `for` apropiado. Observe el cambio en el operador relacional para los ciclos que decrementan la variable de control.

a) Modificar la variable de control de `1` a `100` en incrementos de `1`.

```
for ( int i = 1; i <= 100; i++ )
```

b) Modificar la variable de control de `100` a `1` en decrementos de `1`.

```
for ( int i = 100; i >= 1; i-- )
```

c) Modificar la variable de control de `7` a `77` en incrementos de `7`.

```
for ( int i = 7; i <= 77; i += 7 )
```

d) Modificar la variable de control de `20` a `2` en decrementos de `2`.

```
for ( int i = 20; i >= 2; i -= 2 )
```

e) Modificar la variable de control con la siguiente secuencia de valores: `2, 5, 8, 11, 14, 17, 20`.

```
for ( int i = 2; i <= 20; i += 3 )
```

f) Modificar la variable de control con la siguiente secuencia de valores: `99, 88, 77, 66, 55, 44, 33, 22, 11, 0`.

```
for ( int i = 99; i >= 0; i -= 11 )
```



Error común de programación 5.6

No utilizar el operador relacional apropiado en la condición de continuación de un ciclo que cuente en forma regresiva (como usar `i <= 1` en lugar de `i >= 1` en un ciclo que cuente en forma regresiva hasta llegar a 1) es generalmente un error lógico.

Aplicación: sumar los enteros pares del 2 al 20

Ahora consideremos dos aplicaciones de ejemplo que demuestran usos simples de la instrucción `for`. La aplicación de la figura 5.5 utiliza una instrucción `for` para sumar los enteros pares del 2 al 20 y guardar el resultado en una variable `int` llamada `total`.

```
// Fig. 5.5: Suma.java
// Sumar enteros con la instrucción for.

public class Suma
{
   public static void main( String args[] )
   {
      int total = 0; // inicializa el total

      // total de los enteros pares del 2 al 20
      for ( int numero = 2; numero <= 20; numero += 2 )
         total += numero;

      System.out.printf( “La suma es %d\n”, total ); // muestra los resultados
   } // fin de main
} // fin de la clase Suma
```

```
La suma es 110
```

Figura 5.5 | Sumar enteros con la instrucción `for`.

Las expresiones de *inicialización* e *incremento* pueden ser listas separadas por comas de expresiones que nos permitan utilizar varias expresiones de inicialización, o varias expresiones de incremento. Por ejemplo, aunque esto no se recomienda, el cuerpo de la instrucción `for` en las líneas 11 y 12 de la figura 5.5 podría mezclarse con la porción del incremento del encabezado `for` mediante el uso de una coma, como se muestra a continuación:

```
for ( int numero = 2; numero <= 20; total += numero, numero += 2 )
   ; // instrucción vacía
```



Buena práctica de programación 5.4

Limite el tamaño de los encabezados de las instrucciones de control a una sola línea, si es posible.

Aplicación: cálculo del interés compuesto

La siguiente aplicación utiliza la instrucción `for` para calcular el interés compuesto. Considere el siguiente problema:

> *Una persona invierte $1 000.00 en una cuenta de ahorro que produce el 5% de interés. Suponiendo que todo el interés se deposita en la cuenta, calcule e imprima el monto de dinero en la cuenta al final de cada año, durante 10 años. Use la siguiente fórmula para determinar los montos:*
>
> $c = p(1 + r)^n$
>
> *en donde*
>
> *p* es el monto que se invirtió originalmente (es decir, el monto principal)
> *r* es la tasa de interés anual (por ejemplo, use 0.05 para el 5%)
> *n* es el número de años
> *c* es la cantidad depositada al final del *n*-ésimo año.

Este problema implica el uso de un ciclo que realiza los cálculos indicados para cada uno de los 10 años que el dinero permanece depositado. La solución es la aplicación que se muestra en la figura 5.6. Las líneas 8 a 10 en el método `main` declaran las variables `double` llamadas `monto`, `principal` y `tasa`, e inicializan `principal` con `1000.0` y `tasa` con `0.05`. Java trata a las constantes de punto flotante, como `1000.0` y `0.05`, como de tipo `double`. De manera similar, Java trata a las constantes de números enteros, como `7` y `-22`, como de tipo `int`.

La línea 13 imprime en pantalla los encabezados para las dos columnas de resultados de esta aplicación. La primera columna muestra el año y la segunda, la cantidad depositada al final de ese año. Observe que utilizamos el especificador de formato `%20s` para mostrar en pantalla el objeto `String` `"Monto en deposito"`. El entero `20` después del `%` y el carácter de conversión `s` indica que el valor a imprimir debe mostrarse con una **anchura de campo** de 20; esto es, `printf` debe mostrar el valor con al menos 20 posiciones de caracteres. Si el valor a imprimir tiene una anchura menor a 20 posiciones de caracteres (en este ejemplo son 17 caracteres), el valor se **justifica a la derecha** en el campo de manera predeterminada. Si el valor `anio` a imprimir tuviera una anchura mayor a cuatro posiciones de caracteres, la anchura del campo se extendería a la derecha para dar cabida a todo el valor; esto desplazaría al campo `monto` a la derecha, con lo que se desacomodarían las columnas ordenadas de nuestros resultados tabulares. Para indicar que los valores deben imprimirse **justificados a la izquierda**, sólo hay que anteponer a la anchura de campo la **bandera de formato de signo negativo** (–).

La instrucción `for` (líneas 16 a 23) ejecuta su cuerpo 10 veces, con lo cual la variable de control `anio` varía de 1 a 10, en incrementos de 1. Este ciclo termina cuando la variable de control `anio` se vuelve 11 (observe que `anio` representa a la *n* en el enunciado del problema).

Las clases proporcionan métodos que realizan tareas comunes sobre los objetos. De hecho, la mayoría de los métodos a llamar deben pertenecer a un objeto específico. Por ejemplo, para imprimir texto en la figura 5.6, la línea 13 llama al método `printf` en el objeto `System.out`. Muchas clases también cuentan con métodos que realizan tareas comunes y no requieren objetos. En la sección 3.9 vimos que a estos métodos se les llama `static`. Por ejemplo, Java no incluye un operador de exponenciación, por lo que los diseñadores de la clase `Math` definieron el método `static` llamado `pow` para elevar un valor a una potencia. Para llamar a un método `static` debe especificar el nombre de la clase, seguido de un punto (`.`) y el nombre del método, como en

NombreClase.nombreMétodo (argumentos)

```
1  // Fig. 5.6: Interes.java
2  // Cálculo del interés compuesto con for.
3
4  public class Interes
5  {
6     public static void main( String args[] )
7     {
8        double monto; // Monto depositado al final de cada año
9        double principal = 1000.0; // monto inicial antes de los intereses
10       double tasa = 0.05; // tasa de interés
11
12       // muestra los encabezados
13       System.out.printf( "s%20s\n", "Anio", "Monto en deposito" );
14
15       // calcula el monto en deposito para cada uno de diez años
16       for ( int anio = 1; anio <= 10; anio++ )
17       {
18          // calcula el nuevo monto para el año especificado
19          monto = principal * Math.pow( 1.0 + tasa, anio );
20
21          // muestra el año y el monto
22          System.out.printf( "%4d%,20.2f\n", anio, monto );
23       } // fin de for
24    } // fin de main
25 } // fin de la clase Interes
```

```
Anio   Monto en deposito
   1            1,050.00
   2            1,102.50
   3            1,157.63
   4            1,215.51
   5            1,276.28
   6            1,340.10
   7            1,407.10
   8            1,477.46
   9            1,551.33
  10            1,628.89
```

Figura 5.6 | Cálculo del interés compuesto con `for`.

En el capítulo 6 aprenderá a implementar métodos `static` en sus propias clases.

Utilizamos el método `static pow` de la clase `Math` para realizar el cálculo del interés compuesto en la figura 5.6. `Math.pow(`*x, y*`)` calcula el valor de `x` elevado a la *y*-ésima potencia. El método recibe dos argumentos `double` y devuelve un valor `double`. La línea 19 realiza el cálculo $c = p(1 + r)^n$, en donde c es `monto`, p es `principal`, r es `tasa` y n es `anio`.

Después de cada cálculo, la línea 22 imprime en pantalla el año y el monto depositado al final de ese año. El año se imprime en una anchura de campo de cuatro caracteres (según lo especificado por `%4d`). El monto se imprime como un número de punto flotante con el especificador de formato `%,20.2f`. La **bandera de formato coma** (`,`) indica que el valor debe imprimirse con un separador de agrupamiento. El separador que se utiliza realmente es específico de la configuración regional del usuario (es decir, el país). Por ejemplo, en los Estados Unidos, el número se imprimirá usando comas para separar cada tres dígitos, y un punto decimal para separar la parte fraccionaria del número, como en 1,234.45. El número `20` en la especificación de formato indica que el valor debe imprimirse justificado a la derecha, con una anchura de campo de 20 caracteres. El `.2` especifica la precisión del número con formato; en este caso, el número se redondea a la centésima más cercana y se imprime con dos dígitos a la derecha del punto decimal.

En este ejemplo declaramos las variables `monto`, `capital` y `tasa` de tipo `double`. Estamos tratando con partes fraccionales de dólares y, por ende, necesitamos un tipo que permita puntos decimales en sus valores. Por desgracia, los números de punto flotante pueden provocar problemas. He aquí una sencilla explicación de lo que puede salir mal al utilizar `double` (o `float`) para representar montos en dólares (suponiendo que los montos en dólares se muestran con dos dígitos a la derecha del punto decimal): dos montos en dólares tipo `double` almacenados en la máquina podrían ser 14.234 (que por lo general se redondea a 14.23 para fines de mostrarlo en pantalla) y 18.673 (que por lo general se redondea a 18.67 para fines de mostrarlo en pantalla). Al sumar estos montos, producen una suma interna de 32.907, que por lo general se redondea a 32.91 para fines de mostrarlo en pantalla. Por lo tanto, sus resultados podrían aparecer como

```
  14.23
+ 18.67
-------
  32.91
```

pero una persona que sume los números individuales, como se muestran, esperaría que la suma fuera de 32.90. ¡Ya ha sido advertido!



Buena práctica de programación 5.5

No utilice variables de tipo `double` *(o* `float`*) para realizar cálculos monetarios precisos. La imprecisión de los números de punto flotante puede provocar errores. En los ejercicios usaremos enteros para realizar cálculos monetarios.*

Algunos distribuidores independientes venden bibliotecas de clase que realizan cálculos monetarios precisos. Además, la API de Java cuenta con la clase `java.math.BigDecimal` para realizar cálculos con valores de punto flotante y precisión arbitraria.

Observe que el cuerpo de la instrucción `for` contiene el cálculo `1.0 + tasa`, el cual aparece como argumento para el método `Math.pow`. De hecho, este cálculo produce el mismo resultado cada vez que se realiza una iteración en el ciclo, por lo que repetir el cálculo en todas las iteraciones del ciclo es un desperdicio.



Tip de rendimiento 5.2

En los ciclos, evite cálculos para los cuales el resultado nunca cambia; dichos cálculos, por lo general, deben colocarse antes del ciclo. [Nota: actualmente, muchos de los compiladores con optimización colocan dichos cálculos fuera de los ciclos en el código compilado].

5.5 Instrucción de repetición `do...while`

La instrucción de repetición `do...while` es similar a la instrucción `while`, ya que el programa evalúa la condición de continuación de ciclo al principio, antes de ejecutar el cuerpo del ciclo. Si la condición es falsa, el cuerpo nunca se ejecuta. La instrucción `do...while` evalúa la condición de continuación de ciclo *después* de ejecutar el cuerpo del ciclo; por lo tanto, el cuerpo siempre se ejecuta por lo menos una vez. Cuando termina una instrucción `do...while`, la ejecución continúa con la siguiente instrucción en la secuencia. La figura 5.7 utiliza una instrucción `do...while` para imprimir los números del 1 al 10.

La línea 8 declara e inicializa la variable de control `contador`. Al entrar a la instrucción `do...while`, la línea 12 imprime el valor de `contador` y la 13 incrementa a `contador`. Después, el programa evalúa la prueba de continuación de ciclo al final del mismo (línea 14). Si la condición es verdadera, el ciclo continúa a partir de la primera instrucción del cuerpo en la instrucción `do...while` (línea 12). Si la condición es falsa, el ciclo termina y el programa continúa con la siguiente instrucción después del ciclo.

La figura 5.8 contiene el diagrama de actividad de UML para la instrucción `do...while`. Este diagrama hace evidente que la condición de continuación de ciclo no se evalúa sino hasta después que el ciclo ejecuta el estado de acción, por lo menos una vez. Compare este diagrama de actividad con el de la instrucción `while` (figura 4.4).

No es necesario utilizar llaves en la estructura de repetición `do...while` si sólo hay una instrucción en el cuerpo. Sin embargo, la mayoría de los programadores incluyen las llaves para evitar la confusión entre las instrucciones `while` y `do...while`. Por ejemplo:

`while (` *condición* `)`

```
// Fig. 5.7: PruebaDoWhile.java
// La instrucción de repetición do...while.

public class PruebaDoWhile
{
   public static void main( String args[] )
   {
      int contador = 1; // inicializa contador

      do
      {
         System.out.printf( "%d  ", contador );
         ++contador;
      } while ( contador <= 10 ); // fin de do...while

      System.out.println(); // imprime una nueva línea
   } // fin de main
} // fin de la clase PruebaDoWhile
```

```
1  2  3  4  5  6  7  8  9  10
```

Figura 5.7 | La instrucción de repetición `do...while`.

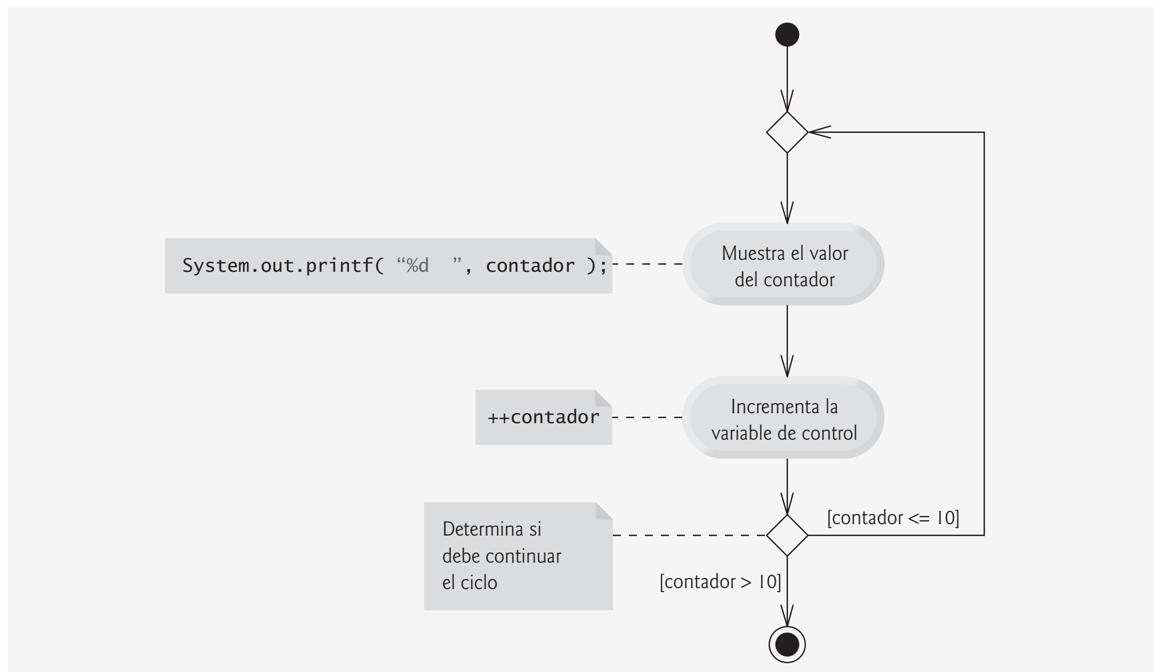


Figura 5.8 | Diagrama de actividad de UML de la instrucción de repetición `do...while`.

generalmente es la primera línea de una instrucción `while`. Una instrucción `do...while` sin llaves, alrededor de un cuerpo con una sola instrucción, aparece así:

```
do
    instrucción
while ( condición );
```

lo cual puede ser confuso. Un lector podría malinterpretar la última línea [`while(` *condición* `);`], como una instrucción `while` que contiene una instrucción vacía (el punto y coma por sí solo). Por ende, la instrucción `do...while` con una instrucción en su cuerpo se escribe generalmente así:

```
do
{
    Instrucción
}  while ( condición );
```



Buena práctica de programación 5.6

Incluya siempre las llaves en una instrucción `do...while`, aun y cuando éstas no sean necesarias. Esto ayuda a eliminar la ambigüedad entre las instrucciones `while` y `do...while` que contienen sólo una instrucción.

5.6 Instrucción de selección múltiple switch

En el capítulo 4 hablamos sobre la instrucción `if` de selección simple y la instrucción `if...else` de selección doble. Java cuenta con la instrucción `switch` de selección múltiple para realizar distintas acciones, con base en los posibles valores de una variable o expresión entera. Cada acción se asocia con un valor de una **expresión entera constante** (es decir, un valor constante de tipo `byte`, `short`, `int` o `char`, pero no `long`) que la variable o expresión en la que se basa la instrucción `switch` pueda asumir.

La clase `LibroCalificaciones` con la instrucción `switch` para contar las calificaciones A, B, C, D y F

La figura 5.9 contiene una versión mejorada de la clase `LibroCalificaciones` que presentamos en el capítulo 3 y desarrollamos un poco más en el capítulo 4. La versión de la clase que presentamos ahora no sólo calcula el promedio de un conjunto de calificaciones numéricas introducidas por el usuario, sino que utiliza una instrucción `switch` para determinar si cada calificación es el equivalente de A, B, C, D o F, y para incrementar el contador de la calificación apropiada. La clase también imprime en pantalla un resumen del número de estudiantes que recibieron cada calificación. La figura 5.10 muestra la entrada y la salida de ejemplo de la aplicación `PruebaLibroCalificaciones`, que utiliza la clase `LibroCalificaciones` para procesar un conjunto de calificaciones.

```
// Fig. 5.9: LibroCalificaciones.java
// La clase LibroCalificaciones usa la instrucción switch para contar las calificaciones
A, B, C, D y F.
import java.util.Scanner; // el programa usa la clase Scanner

public class LibroCalificaciones
{
   private String nombreDelCurso; // nombre del curso que representa este
   LibroCalificaciones
   private int total; // suma de las calificaciones
   private int contadorCalif; // número de calificaciones introducidas
   private int aCuenta; // cuenta de calificaciones A
   private int bCuenta; // cuenta de calificaciones B
   private int cCuenta; // cuenta de calificaciones C
   private int dCuenta; // cuenta de calificaciones D
   private int fCuenta; // cuenta de calificaciones F

   // el constructor inicializa nombreDelCurso;
   // las variables de instancia int se inicializan en 0 de manera predeterminada
   public LibroCalificaciones( String nombre )
   {
```

Figura 5.9 | Clase `LibroCalificaciones` que utiliza una instrucción `switch` para contar las calificaciones A, B, C, D y F. (Parte 1 de 3).

```
      nombreDelCurso = nombre; // inicializa nombreDelCurso
   } // fin del constructor

   // método para establecer el nombre del curso
   public void establecerNombreDelCurso( String nombre )
   {
      nombreDelCurso = nombre; // almacena el nombre del curso
   } // fin del método establecerNombreDelCurso

   // método para obtener el nombre del curso
   public String obtenerNombreDelCurso()
   {
      return nombreDelCurso;
   } // fin del método obtenerNombreDelCurso

   // muestra un mensaje de bienvenida al usuario de LibroCalificaciones
   public void mostrarMensaje()
   {
      // obtenerNombreDelCurso obtiene el nombre del curso
      System.out.printf( "Bienvenido al libro de calificaciones para\n%s!\n\n",
         obtenerNombreDelCurso() );
   } // fin del método mostrarMensaje

   // introduce un número arbitrario de calificaciones del usuario
   public void introducirCalif()
   {
      Scanner entrada = new Scanner( System.in );

      int calificacion; // calificación introducida por el usuario

      System.out.printf( "%s\n%s\n   %s\n   %s\n",
         "Escriba las calificaciones enteras en el rango de 0 a 100.",
         "Escriba el indicador de fin de archivo para terminar la entrada:",
         "En UNIX/Linux/Mac OS X escriba <ctrl> d y después oprima Intro",
         "En Windows escriba <ctrl> z y después oprima Intro" );

      // itera hasta que el usuario introduzca el indicador de fin de archivo
      while ( entrada.hasNext() )
      {
         calificacion = entrada.nextInt(); // lee calificación
         total += calificacion; // suma calificación a total
         ++contadorCalif; // incrementa el número de calificaciones

         // llama al método para incrementar el contador apropiado
         incrementarContadorCalifLetra( calificacion );
      } // fin de while
   } // fin del método introducirCalif

   // suma 1 al contador apropiado para la calificación especificada
   public void incrementarContadorCalifLetra( int calificacion )
   {
      // determina cuál calificación se introdujo
      switch ( calificacion / 10 )
      {
         case 9:  // calificación está entre 90
         case 10: // y 100
           ++aCuenta; // incrementa aCuenta
           break; // necesaria para salir del switch
```

Figura 5.9 | Clase `LibroCalificaciones` que utiliza una instrucción `switch` para contar las calificaciones A, B, C, D y F. (Parte 2 de 3).

```
78
79            case 8: // calificación está entre 80 y 89
80              ++bCuenta; // incrementa bCuenta
81              break; // sale del switch
82
83            case 7: // calificación está entre 70 y 79
84              ++cCuenta; // incrementa cCuenta
85              break; // sale del switch
86
87            case 6: // calificación está entre 60 y 69
88              ++dCuenta; // incrementa dCuenta
89              break; // sale del switch
90
91            default: // calificación es menor que 60
92              ++fCuenta; // incrementa fCuenta
93              break; // opcional; de todas formas sale del switch
94         } // fin de switch
95      } // fin del método incrementarContadorCalifLetra
96
97      // muestra un reporte con base en las calificaciones introducidas por el usuario
98      public void mostrarReporteCalif()
99      {
100        System.out.println( "\nReporte de calificaciones:" );
101
102        // si el usuario introdujo por lo menos una calificación...
103        if ( contadorCalif != 0 )
104        {
105           // calcula el promedio de todas las calificaciones introducidas
106           double promedio = (double) total / contadorCalif;
107
108           // imprime resumen de resultados
109           System.out.printf( "El total de las %d calificaciones introducidas es %d\n",
110             contadorCalif, total );
111           System.out.printf( "El promedio de la clase es %.2f\n", promedio );
112           System.out.printf( "%s\n%s%d\n%s%d\n%s%d\n%s%d\n%s%d\n",
113             "Numero de estudiantes que recibieron cada calificacion:",
114             "A: ", aCuenta,   // muestra el número de calificaciones A
115             "B: ", bCuenta,   // muestra el número de calificaciones B
116             "C: ", cCuenta,   // muestra el número de calificaciones C
117             "D: ", dCuenta,   // muestra el número de calificaciones D
118             "F: ", fCuenta ); // muestra el número de calificaciones F
119        } // fin de if
120        else // no se introdujeron calificaciones, por lo que imprime el mensaje
             apropiado
121           System.out.println( "No se introdujeron calificaciones" );
122     } // fin del método mostrarReporteCalif
123  } // fin de la clase LibroCalificaciones
```

Figura 5.9 | Clase `LibroCalificaciones` que utiliza una instrucción `switch` para contar las calificaciones A, B, C, D y F. (Parte 3 de 3).

Al igual que las versiones anteriores de la clase, `LibroCalificaciones` (figura 5.9) declara la variable de instancia `nombreDelCurso` (línea 7) y contiene los métodos `establecerNombreDelCurso` (líneas 24 a 27), `obtenerNombreDelCurso` (líneas 30 a 33) y `mostrarMensaje` (líneas 36 a 41), que establecen el nombre del curso, lo almacenan y muestran un mensaje de bienvenida al usuario, respectivamente. La clase también contiene un constructor (líneas 18 a 21) que inicializa el nombre del curso.

La clase `LibroCalificaciones` también declara las variables de instancia `total` (línea 8) y `contadorCalif` (línea 9), que llevan la cuenta de la suma de las calificaciones introducidas por el usuario y el número de cali-

ficaciones introducidas, respectivamente. Las líneas 10 a 14 declaran las variables contador para cada categoría de calificaciones. La clase `LibroCalificaciones` mantiene a `total`, `contadorCalif` y a los cinco contadores de las letras de calificación como variables de instancia, de manera que estas variables puedan utilizarse o modificarse en cualquiera de los métodos de la clase. Observe que el constructor de la clase (líneas 18 a 21) establece sólo el nombre del curso; las siete variables de instancia restantes son de tipo `int` y se inicializan con `0`, de manera predeterminada.

La clase `LibroCalificaciones` (figura 5.9) contiene tres métodos adicionales: `introducirCalif`, `incrementarContadorCalifLetra` y `mostrarReporteCalif`. El método `introducirCalif` (líneas 44 a 66) lee un número arbitrario de calificaciones enteras del usuario mediante el uso de la repetición controlada por un centinela, y actualiza las variables de instancia `total` y `contadorCalif`. El método `introducirCalif` llama al método `incrementarContadorCalifLetra` (líneas 69 a 95) para actualizar el contador de letra de calificación apropiado para cada calificación introducida. La clase `LibroCalificaciones` también contiene el método `mostrarReporteCalif` (líneas 98 a 122), el cual imprime en pantalla un reporte que contiene el total de todas las calificaciones introducidas, el promedio de las mismas y el número de estudiantes que recibieron cada letra de calificación. Examinaremos estos métodos con más detalle.

La línea 48 en el método `introducirCalif` declara la variable `calificacion` que almacenará la entrada del usuario. Las líneas 50 a 54 piden al usuario que introduzca calificaciones enteras y escriba el indicador de fin de archivo para terminar la entrada. El **indicador de fin de archivo** es una combinación de teclas dependiente del sistema, que el usuario introduce para indicar que no hay más datos que introducir. En el capítulo 14, Archivos y flujos, veremos cómo se utiliza el indicador de fin de archivo cuando un programa lee su entrada desde un archivo.

En los sistemas UNIX/Linux/Mac OS X, el fin de archivo se introduce escribiendo la secuencia

<ctrl> d

en una línea por sí sola. Esta notación significa que hay que oprimir al mismo tiempo la tecla *ctrl* y la tecla *d*. En los sistemas Windows, para introducir el fin de archivo se escribe

<ctrl> z

[*Nota:* en algunos sistemas, es necesario oprimir *Intro* después de escribir la secuencia de teclas de fin de archivo. Además, Windows generalmente muestra los caracteres `^Z` en la pantalla cuando se escribe el indicador de fin de archivo, como se muestra en la salida de la figura 5.10].



Tip de portabilidad 5.1

Las combinaciones de teclas para introducir el fin de archivo son dependientes del sistema.

La instrucción `while` (líneas 57 a 65) obtiene la entrada del usuario. La condición en la línea 57 llama al método `hasNext` de `Scanner` para determinar si hay más datos a introducir. Este método devuelve el valor `boolean true` si hay más datos; en caso contrario, devuelve `false`. Después, el valor devuelto se utiliza como el valor de la condición en la instrucción `while`. Mientras no se haya escrito el indicador de fin de archivo, el método `hasNext` devolverá `true`.

La línea 59 recibe como entrada un valor del usuario. La línea 60 utiliza el operador `+=` para sumar `calificacion` a `total`. La línea 61 incrementa `contadorCalif`. El método `mostrarReporteCalif` de la clase utiliza estas variables para calcular el promedio de las calificaciones. La línea 64 llama al método `incrementarContadorCalifLetra` de la clase (declarado en las líneas 69 a 95) para incrementar el contador de letra de calificación apropiado, con base en la calificación numérica introducida.

El método `incrementarContadoraCalifLetra` contiene una instrucción `switch` (líneas 72 a 94) que determina cuál contador se debe incrementar. En este ejemplo, suponemos que el usuario introduce una calificación válida en el rango de 0 a 100. Una calificación en el rango de 90 a 100 representa la A: de 80 a 89, la B; de 70 a 79, la C; de 60 a 69, la D y de 0 a 59, la F. La instrucción `switch` consiste en un bloque que contiene una secuencia de **etiquetas** `case` y una instrucción `case` **default** opcional. Estas etiquetas se utilizan en este ejemplo para determinar cuál contador se debe incrementar, con base en la calificación.

Cuando el flujo de control llega al `switch`, el programa evalúa la expresión entre paréntesis (`calificacion / 10`) que va después de la palabra clave `switch`. A esto se le conoce como la **expresión de control** de la instrucción `switch`. El programa compara el valor de la expresión de control (que se debe evaluar como un valor entero de

tipo byte, char, short o int) con cada una de las etiquetas case. La expresión de control de la línea 72 realiza la división entera, que trunca la parte fraccionaria del resultado. Por ende, cuando dividimos cualquier valor en el rango de 0 a 100 entre 10, el resultado es siempre un valor de 0 a 10. Utilizamos varios de estos valores en nuestras etiquetas case. Por ejemplo, si el usuario introduce el entero 85, la expresión de control se evalúa como el valor int 8. La instrucción switch compara a 8 con cada etiqueta case. Si ocurre una coincidencia (case 8: en la línea 79), el programa ejecuta las instrucciones para esa instrucción case. Para el entero 8, la línea 80 incrementa a bCuenta, ya que una calificación entre 80 y 89 es una B. La instrucción break (línea 81) hace que el control del programa proceda con la primera instrucción después del switch; en este programa, llegamos al final del cuerpo del método incrementarContadorCalifLetra, por lo que el control regresa a la línea 65 en el método introducirCalif (la primera línea después de la llamada a incrementarContadorCalifLetra). Esta línea marca el fin del cuerpo del ciclo while que recibe las calificaciones de entrada (líneas 57 a 65), por lo que el control fluye hacia la condición del while (línea 57) para determinar si el ciclo debe seguir ejecutándose.

Las etiquetas case en nuestro switch evalúan explícitamente los valores 10, 9, 8, 7 y 6. Observe los casos en las líneas 74 y 75, que evalúan los valores 9 y 10 (los cuales representan la calificación A). Al listar las etiquetas case en forma consecutiva, sin instrucciones entre ellas, pueden ejecutar el mismo conjunto de instrucciones; cuando la expresión de control se evalúa como 9 o 10, se ejecutan las instrucciones de las líneas 76 y 77. La instrucción switch no cuenta con un mecanismo para evaluar rangos de valores, por lo que cada valor que deba evaluarse se tiene que listar en una etiqueta case separada. Observe que cada case puede tener varias instrucciones. La instrucción switch es distinta de otras instrucciones de control, en cuanto a que no requiere llaves alrededor de varias instrucciones en cada case.

Sin instrucciones break, cada vez que ocurre una coincidencia en el switch, se ejecutan las instrucciones para ese case y los subsiguientes, hasta encontrar una instrucción break o el final del switch. A menudo a esto se le conoce como que las etiquetas case "se pasarían" hacia las instrucciones en las etiquetas case subsiguientes. (Esta característica es perfecta para escribir un programa conciso, que muestre la canción iterativa "Los Doce Días de Navidad" en el ejercicio 5.29).



Error común de programación 5.7

Olvidar una instrucción break cuando se necesita una en una instrucción switch es un error lógico.

Si no ocurre una coincidencia entre el valor de la expresión de control y una etiqueta case, se ejecuta el caso default (líneas 91 a 93). Utilizamos el caso default en este ejemplo para procesar todos los valores de la expresión de control que sean menores de 6; esto es, todas las calificaciones de reprobado. Si no ocurre una coincidencia y la instrucción switch no contiene un caso default, el control del programa simplemente continúa con la primera instrucción después de la instrucción switch.

La clase **PruebaLibroCalificaciones** *para demostrar la clase* **LibroCalificaciones**

La clase PruebaLibroCalificaciones (figura 5.10) crea un objeto LibroCalificaciones (líneas 10 y 11). La línea 13 invoca el método mostrarMensaje del objeto para imprimir en pantalla un mensaje de bienvenida para el usuario. La línea 14 invoca el método introducirCalif del objeto para leer un conjunto de calificaciones del usuario y llevar el registro de la suma de todas las calificaciones introducidas, y el número de calificaciones. Recuerde que el método introducirCalif también llama al método incrementarContadorCalifLetra para llevar el registro del número de estudiantes que recibieron cada letra de calificación. La línea 15 invoca el método mostrarReporteCalif de la clase LibroCalificaciones, el cual imprime en pantalla un reporte con base en las calificaciones introducidas (como en la ventana de entrada/salida en la figura 5.10). La línea 103 de la clase LibroCalificaciones (figura 5.9) determina si el usuario introdujo por lo menos una calificación; esto evita la división entre cero. De ser así, la línea 106 calcula el promedio de las calificaciones. A continuación, las líneas 109 a 118 imprimen en pantalla el total de todas las calificaciones, el promedio de la clase y el número de estudiantes que recibieron cada letra de calificación. Si no se introdujeron calificaciones, la línea 121 imprime en pantalla un mensaje apropiado. Los resultados en la figura 5.10 muestran un reporte de calificaciones de ejemplo, con base en 10 calificaciones.

Observe que la clase PruebaLibroCalificaciones (figura 5.10) no llama directamente al método incrementarContadorCalifLetra de LibroCalificaciones (líneas 69 a 95 de la figura 5.9). Este método lo utiliza exclusivamente el método introducirCalif de la clase LibroCalificaciones para actualizar el contador de la

calificación de letra apropiado, a medida que el usuario introduce cada nueva calificación. El método `incrementarContadorCalifLetra` existe únicamente para dar soporte a las operaciones de los demás métodos de la clase `LibroCalificaciones`, por lo cual se declara como `private`. En el capítulo 3 vimos que los métodos que se declaran con el modificador de acceso `private` pueden llamarse sólo por otros métodos de la clase en la que están declarados los métodos `private`. Dichos métodos se conocen comúnmente como **métodos utilitarios** o **métodos ayudantes**, debido a que sólo pueden llamarse mediante otros métodos de esa clase y se utilizan para dar soporte a la operación de esos métodos.

```
// Fig. 5.10: PruebaLibroCalificaciones.java
// Crea un objeto LibroCalificaciones, introduce las calificaciones y muestra un reporte.

public class PruebaLibroCalificaciones
{
   public static void main( String args[] )
   {
      // crea un objeto LibroCalificaciones llamado miLibroCalificaciones y
      // pasa el nombre del curso al constructor
      LibroCalificaciones miLibroCalificaciones = new LibroCalificaciones(
         "CS101 Introducción a la programación en Java" );

      miLibroCalificaciones.mostrarMensaje(); // muestra un mensaje de bienvenida
      miLibroCalificaciones.introducirCalif(); // lee calificaciones del usuario
      miLibroCalificaciones.mostrarReporteCalif(); // muestra reporte basado en las calificaciones
   } // fin de main
} // fin de la clase PruebaLibroCalificaciones
```

```
Bienvenido al libro de calificaciones para
CS101 Introduccion a la programacion en Java!

Escriba las calificaciones enteras en el rango de 0 a 100.
Escriba el indicador de fin de archivo para terminar la entrada:
   En UNIX/Linux/Mac OS X escriba <ctrl> d y despues oprima Intro
   En Windows escriba <ctrl> z y despues oprima Intro
99
92
45
57
63
71
76
85
90
100
^Z

Reporte de calificaciones:
El total de las 10 calificaciones introducidas es 778
El promedio de la clase es 77.80
Numero de estudiantes que recibieron cada calificacion:
A: 4
B: 1
C: 2
D: 1
F: 2
```

Figura 5.10 | `PruebaLibroCalificaciones` crea un objeto `LibroCalificaciones` e invoca a sus métodos.

Diagrama de actividad de UML de la instrucción `switch`

La figura 5.11 muestra el diagrama de actividad de UML para la instrucción `switch` general. La mayoría de las instrucciones `switch` utilizan una instrucción `break` en cada `case` para terminar la instrucción `switch` después de procesar el `case`. La figura 5.11 enfatiza esto al incluir instrucciones `break` en el diagrama de actividad. Este diagrama hace evidente que `break` al final de una etiqueta `case` hace que el control salga de la instrucción `switch` de inmediato.

No se requiere una instrucción `break` para la última etiqueta `case` del `switch` (o para el caso `default` opcional, cuando aparece al último), ya que la ejecución continúa con la siguiente instrucción que va después del `switch`.



Observación de ingeniería de software 5.2

Proporcione un caso `default` en las instrucciones `switch`. Al incluir un caso `default` usted puede enfocarse en la necesidad de procesar las condiciones excepcionales.



Buena práctica de programación 5.7

Aunque cada `case` y el caso `default` en una instrucción `switch` pueden ocurrir en cualquier orden, es conveniente colocar la etiqueta `default`. Cuando el caso `default` se lista al último, no se requiere el `break` para ese caso. Algunos programadores incluyen este `break` para mejorar la legibilidad y tener simetría con los demás casos.

Cuando utilice la instrucción `switch`, recuerde que la expresión después de cada `case` debe ser una expresión entera constante; es decir, cualquier combinación de constantes enteras que se evalúen como un valor entero constante (por ejemplo, –7, 0 o 221). Una constante entera es tan solo un valor entero. Además, puede utilizar **constantes tipo carácter**: caracteres específicos entre comillas sencillas, como `'A'`, `'7'` o `'$'`, las cuales

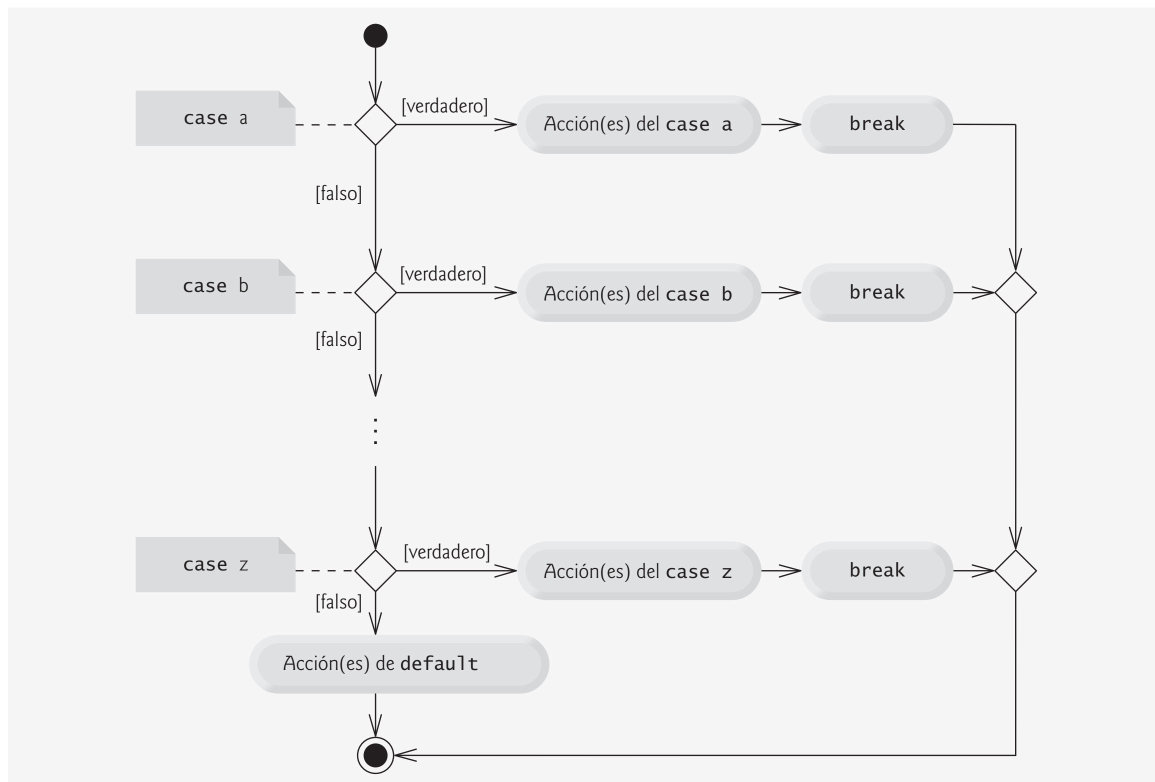


Figura 5.11 | Diagrama de actividad de UML de la instrucción `switch` de selección múltiple con instrucciones `break`.

representan los valores enteros de los caracteres. (En el apéndice B, Conjunto de caracteres ASCII, se muestran los valores enteros de los caracteres en el conjunto de caracteres ASCII, que es un subconjunto del conjunto de caracteres Unicode utilizado por Java).

La expresión en cada `case` también puede ser una **variable constante**: una variable que contiene un valor que no cambia durante todo el programa. Dicha variable se declara mediante la palabra clave `final` (que describiremos en el capítulo 6, Métodos: un análisis más detallado). Java tiene una característica conocida como enumeraciones, que también presentaremos en el capítulo 6. Las constantes de enumeración también pueden utilizarse en etiquetas `case`. En el capítulo 10, Programación orientada a objetos: polimorfismo, presentaremos una manera más elegante de implementar la lógica del `switch`; utilizaremos una técnica llamada polimorfismo para crear programas que a menudo son más legibles, fáciles de mantener y de extender que los programas que utilizan lógica de `switch`.

5.7 Instrucciones break y continue

Además de las instrucciones de selección y repetición, Java cuenta con las instrucciones `break` y `continue` (que presentamos en esta sección y en el apéndice N, Instrucciones `break` y `continue` etiquetadas) para alterar el flujo de control. En la sección anterior mostramos cómo puede utilizarse la instrucción `break` para terminar la ejecución de una instrucción `switch`. En esta sección veremos cómo utilizar `break` en las instrucciones de repetición.

Java también cuenta con las instrucciones `break` y `continue` etiquetadas, para usarlas en los casos en los que es conveniente alterar el flujo de control en las instrucciones de control anidadas. En el apéndice N hablaremos sobre las instrucciones `break` y `continue` etiquetadas.

Instrucción break

Cuando `break` se ejecuta en una instrucción `while`, `for`, `do...while`, o `switch`, ocasiona la salida inmediata de esa instrucción. La ejecución continúa con la primera instrucción después de la instrucción de control. Los usos comunes de `break` son para escapar anticipadamente del ciclo, o para omitir el resto de una instrucción `switch` (como en la figura 5.9). La figura 5.12 demuestra el uso de una instrucción `break` para salir de un ciclo `for`.

```
// Fig. 5.12: PruebaBreak.java
// La instrucción break para salir de una instrucción for.
public class PruebaBreak
{
   public static void main( String args[] )
   {
      int cuenta; // la variable de control también se usa cuando termina el ciclo

      for ( cuenta = 1; cuenta <= 10; cuenta++ ) // itera 10 veces
      {
         if ( cuenta == 5 ) // si cuenta es 5,
            break;     // termina el ciclo

         System.out.printf( "%d ", cuenta );
      } // fin de for

      System.out.printf( "\nSalio del ciclo en cuenta = %d\n", cuenta );
   } // fin de main
} // fin de la clase PruebaBreak
```

```
1 2 3 4
Salio del ciclo en cuenta = 5
```

Figura 5.12 | La instrucción `break` para salir de una instrucción `for`.

Cuando la instrucción `if` anidada en la línea 11 dentro de la instrucción `for` (líneas 9 a 15) determina que `cuenta` es `5`, se ejecuta la instrucción `break` en la línea 12. Esto termina la instrucción `for` y el programa continúa a la línea 17 (inmediatamente después de la instrucción `for`), la cual muestra un mensaje indicando el valor de la variable de control cuando terminó el ciclo. El ciclo ejecuta su cuerpo por completo sólo cuatro veces en vez de 10.

Instrucción `continue`

Cuando la instrucción `continue` se ejecuta en una instrucción `while`, `for` o `do...while`, omite las instrucciones restantes en el cuerpo del ciclo y continúa con la siguiente iteración del ciclo. En las instrucciones `while` y `do...while`, la aplicación evalúa la prueba de continuación de ciclo justo después de que se ejecuta la instrucción `continue`. En una instrucción `for` se ejecuta la expresión de incremento y después el programa evalúa la prueba de continuación de ciclo.

La figura 5.13 utiliza la instrucción `continue` en un ciclo `for` para omitir la instrucción de la línea 12 cuando la instrucción `if` anidada (línea 9) determina que el valor de `cuenta` es `5`. Cuando se ejecuta la instrucción `continue`, el control del programa continúa con el incremento de la variable de control en la instrucción `for` (línea 7).

En la sección 5.3 declaramos que la instrucción `while` puede utilizarse, en la mayoría de los casos, en lugar de `for`. La única excepción ocurre cuando la expresión de incremento en el `while` va después de una instrucción `continue`. En este caso, el incremento no se ejecuta antes de que el programa evalúe la condición de continuación de repetición, por lo que el `while` no se ejecuta de la misma manera que el `for`.



Observación de ingeniería de software 5.3

Algunos programadores sienten que las instrucciones `break` *y* `continue` *violan la programación estructurada. Ya que pueden lograrse los mismos efectos con las técnicas de programación estructurada, estos programadores prefieren no utilizar instrucciones* `break` *o* `continue`*.*



Observación de ingeniería de software 5.4

Existe una tensión entre lograr la ingeniería de software de calidad y lograr el software con mejor desempeño. A menudo, una de estas metas se logra a expensas de la otra. Para todas las situaciones excepto las que demanden el mayor rendimiento, aplique la siguiente regla empírica: primero, asegúrese de que su código sea simple y correcto; después hágalo rápido y pequeño, pero sólo si es necesario.

```
1   // Fig. 5.13: PruebaContinue.java
2   // Instrucción continue para terminar una iteración de una instrucción for.
3   public class PruebaContinue
4   {
5      public static void main( String args[] )
6      {
7         for ( int cuenta = 1; cuenta <= 10; cuenta++ ) // itera 10 veces
8         {
9            if ( cuenta == 5 ) // si cuenta es 5,
10              continue; // omite el resto del código en el ciclo
11
12           System.out.printf( "%d ", cuenta );
13        } // fin de for
14
15        System.out.println( "\nSe uso continue para omitir imprimir 5" );
16     } // fin de main
17  } // fin de la clase PruebaContinue
```

```
1 2 3 4 6 7 8 9 10
Se uso continue para omitir imprimir 5
```

Figura 5.13 | Instrucción `continue` para terminar una iteración de una instrucción `for`.

5.8 Operadores lógicos

Cada una de las instrucciones `if`, `if...else`, `while`, `do...while` y `for` requieren una condición para determinar cómo continuar con el flujo de control de un programa. Hasta ahora sólo hemos estudiado las **condiciones simples**, como `cuenta <= 10`, `numero != valorCentinela` y `total > 1000`. Las condiciones simples se expresan en términos de los operadores relacionales `>`, `<`, `>=` y `<=`, y los operadores de igualdad `==` y `!=`; cada expresión evalúa sólo una condición. Para evaluar condiciones múltiples en el proceso de tomar una decisión, ejecutamos estas pruebas en instrucciones separadas o en instrucciones `if` o `if...else` anidadas. En ocasiones, las instrucciones de control requieren condiciones más complejas para determinar el flujo de control de un programa.

Java cuenta con los **operadores lógicos** para que usted pueda formar condiciones más complejas, al combinar las condiciones simples. Los operadores lógicos son `&&` (AND condicional), `||` (OR condicional), `&` (AND lógico booleano), `|` (OR inclusivo lógico booleano), `^` (OR exclusivo lógico booleano) y `!` (NOT lógico).

Operador AND (&&) condicional

Suponga que deseamos asegurar en cierto punto de una aplicación que dos condiciones sean *ambas* verdaderas, antes de elegir cierta ruta de ejecución. En este caso, podemos utilizar el operador **&&** (**AND condicional**) de la siguiente manera:

```
if ( genero == FEMENINO && edad >= 65 )
   ++mujeresMayores;
```

Esta instrucción `if` contiene dos condiciones simples. La condición `genero == FEMENINO` compara la variable `genero` con la constante `FEMENINO`. Por ejemplo, esto podría evaluarse para determinar si una persona es mujer. La condición `edad >= 65` podría evaluarse para determinar si una persona es un ciudadano mayor. La instrucción `if` considera la condición combinada

```
genero == FEMENINO && edad >= 65
```

la cual es verdadera si, y sólo si ambas condiciones simples son verdaderas. Si la condición combinada es verdadera, el cuerpo de la instrucción `if` incrementa a `mujeresMayores` en `1`. Si una o ambas condiciones simples son falsas, el programa omite el incremento. Algunos programadores consideran que la condición combinada anterior es más legible si se agregan paréntesis redundantes, como por ejemplo:

```
( genero == FEMENINO ) && ( edad >= 65 )
```

La tabla de la figura 5.14 sintetiza el uso del operador `&&`. Esta tabla muestra las cuatro combinaciones posibles de valores `false` y `true` para *expresión1* y *expresión2*. A dichas tablas se les conoce como **tablas de verdad**. Java evalúa todas las expresiones que incluyen operadores relacionales, de igualdad o lógicos como `true` o `false`.

expresión1	expresión2	expresión1 && expresión2
false	false	false
false	true	false
true	false	false
true	true	true

Figura 5.14 | Tabla de verdad del operador `&&` (AND condicional).

Operador OR condicional (||)

Ahora suponga que deseamos asegurar que cualquiera *o* ambas condiciones sean verdaderas antes de elegir cierta ruta de ejecución. En este caso, utilizamos el operador `||` (**OR condicional**), como se muestra en el siguiente segmento de un programa:

```
if ( ( promedioSemestre >= 90 ) || ( examenFinal >= 90 ) )
   System.out.println ( "La calificacion del estudiante es A" );
```

Esta instrucción también contiene dos condiciones simples. La condición `promedioSemestre >= 90` se evalúa para determinar si el estudiante merece una A en el curso, debido a que tuvo un sólido rendimiento a lo largo del semestre. La condición `examenFinal >= 90` se evalúa para determinar si el estudiante merece una A en el curso debido a un desempeño sobresaliente en el examen final. Después, la instrucción `if` considera la condición combinada

```
( promedioSemestre >= 90 ) || ( examenFinal >= 90 )
```

y otorga una A al estudiante si una o ambas de las condiciones simples son verdaderas. La única vez que *no* se imprime el mensaje "`La calificación del estudiante es A`" es cuando ambas condiciones simples son falsas. La figura 5.15 es una tabla de verdad para el operador OR condicional (`||`). El operador `&&` tiene mayor precedencia que el operador `||`. Ambos operadores se asocian de izquierda a derecha.

expresión1	expresión2	expresión1 || expresión2
`false`	`false`	`false`
`false`	`true`	`true`
`true`	`false`	`true`
`true`	`true`	`true`

Figura 5.15 | Tabla de verdad del operador (OR condicional) `||`.

Evaluación en corto circuito de condiciones complejas

Las partes de una expresión que contienen los operadores `&&` o `||` se evalúan sólo hasta que se sabe si la condición es verdadera o falsa. Por ende, la evaluación de la expresión

```
( genero == FEMENINO ) && ( edad >= 65 )
```

se detiene de inmediato si `genero` no es igual a `FEMENINO` (es decir, en este punto es evidente que toda la expresión es `false`) y continúa si `genero` *es* igual a `FEMENINO` (es decir, toda la expresión podría ser aún `true` si la condición `edad >= 65` es `true`). Esta característica de las expresiones AND y OR condicional se conoce como **evaluación en corto circuito**.



Error común de programación 5.8

En las expresiones que utilizan el operador `&&`, una condición (a la cual le llamamos condición dependiente) puede requerir que otra condición sea verdadera para que la evaluación de la condición dependiente tenga significado. En este caso, la condición dependiente debe colocarse después de la otra condición, o podría ocurrir un error. Por ejemplo, en la expresión `( i != 0) && (10 / i == 2)`, la segunda condición debe aparecer después de la primera, o podría ocurrir un error de división entre cero.

Operadores AND lógico booleano (&) y OR inclusivo lógico booleano (|)

Los operadores **AND lógico booleano** (`&`) y **OR inclusivo lógico booleano** (`|`) funcionan en forma idéntica a los operadores `&&` (AND condicional) y `||` (OR condicional), con una excepción: los operadores lógicos booleanos siempre evalúan ambos operandos (es decir, no realizan una evaluación en corto circuito). Por lo tanto, la expresión

```
( genero == 1 ) & ( edad >= 65 )
```

evalúa `edad >= 65`, sin importar que `genero` sea igual o no a `1`. Esto es útil si el operando derecho del operador AND lógico booleano o del OR inclusivo lógico booleano tiene un **efecto secundario** requerido: la modificación del valor de una variable. Por ejemplo, la expresión

```
( cumpleanios == true ) | ( ++edad >= 65 )
```

garantiza que se evalúe la condición `++edad >= 65`. Por ende, la variable `edad` se incrementa en la expresión anterior, sin importar que la expresión total sea `true` o `false`.



Tip para prevenir errores 5.4

Por cuestión de claridad, evite las expresiones con efectos secundarios en las condiciones. Los efectos secundarios pueden tener una apariencia inteligente, pero pueden hacer que el código sea más difícil de entender y pueden llegar a producir errores lógicos sutiles.

OR exclusivo lógico booleano (^)

Una condición compleja que contiene el operador **OR exclusivo lógico booleano** (^) es `true` *si y sólo si uno de sus operandos es `true` y el otro es `false`*. Si ambos operandos son `true` o si ambos son `false`, toda la condición es `false`. La figura 5.16 es una tabla de verdad para el operador OR exclusivo lógico booleano (^). También se garantiza que este operador evaluará ambos operandos.

expresión1	expresión2	expresión1 ^ expresión2
false	false	false
false	true	true
true	false	true
true	true	false

Figura 5.16 | Tabla de verdad del operador ^ (OR exclusivo lógico booleano).

Operador lógico de negación (!)

El operador **!** (**NOT lógico**, también conocido como **negación lógica** o **complemento lógico**) "invierte" el significado de una condición. A diferencia de los operadores lógicos &&, ||, &, | y ^, que son operadores binarios que combinan dos condiciones, el operador lógico de negación es un operador unario que sólo tiene una condición como operando. Este operador se coloca antes de una condición para elegir una ruta de ejecución si la condición original (sin el operador lógico de negación) es `false`, como en el siguiente segmento de código:

```
if ( ! ( calificacion == valorCentinela ) )
   System.out.printf( "La siguiente calificación es %d\n", calificacion );
```

que ejecuta la llamada a `printf` sólo si `calificacion` no es igual a `valorCentinela`. Los paréntesis alrededor de la condición `calificacion == valorCentinela` son necesarios, ya que el operador lógico de negación tiene mayor precedencia que el de igualdad.

En la mayoría de los casos, puede evitar el uso de la negación lógica si expresa la condición en forma distinta, con un operador relacional o de igualdad apropiado. Por ejemplo, la instrucción anterior también puede escribirse de la siguiente manera:

```
if ( calificacion != valorCentinela )
   System.out.printf( "La siguiente calificación es %d\n", calificacion );
```

Esta flexibilidad le puede ayudar a expresar una condición de una manera más conveniente. La figura 5.17 es una tabla de verdad para el operador lógico de negación.

expresión	!expresión
false	true
true	false

Figura 5.17 | Tabla de verdad del operador ! (negación lógica, o NOT lógico).

Ejemplo de los operadores lógicos

La figura 5.18 demuestra el uso de los operadores lógicos y lógicos booleanos; para ello produce sus tablas de verdad. Los resultados muestran la expresión que se evalúo y el resultado `boolean` de esa expresión. Los valores de las expresiones `boolean` se muestran mediante `printf`, usando el **especificador de formato %b**, que imprime la palabra "true" o "false", con base en el valor de la expresión. Las líneas 9 a 13 producen la tabla de verdad para el `&&`. Las líneas 16 a 20 producen la tabla de verdad para el `||`. Las líneas 23 a 27 producen la tabla de verdad para el `&`. Las líneas 30 a 35 producen la tabla de verdad para el `|`. Las líneas 38 a 43 producen la tabla de verdad para el `^`. Las líneas 46 a 47 producen la tabla de verdad para el `!`.

```
1  // Fig. 5.18: OperadoresLogicos.java
2  // Los operadores lógicos.
3
4  public class OperadoresLogicos
5  {
6     public static void main( String args[] )
7     {
8        // crea tabla de verdad para el operador && (AND condicional)
9        System.out.printf( "s\n%s: %b\n%s: %b\n%s: %b\n%s: %b\n\n",
10          "AND condicional (&&)", "false && false"( false && false ),
11          "false && true", ( false && true ),
12          "true && false", ( true && false ),
13          "true && true", ( true && true ) );
14
15       // crea tabla de verdad para el operador || (OR condicional)
16       System.out.printf( "%s\n%s: %b\n%s: %b\n%s: %b\n%s: %b\n\n",
17          "OR condicional (||)", "false || false", ( false || false ),
18          "false || true", ( false || true ),
19          "true || false", ( true || false ),
20          "true || true", ( true || true ) );
21
22       // crea tabla de verdad para el operador & (AND lógico booleano)
23       System.out.printf( "%s\n%s: %b\n%s: %b\n%s: %b\n%s: %b\n\n",
24          "AND logico booleano (&)", "false & false", ( false & false ),
25          "false & true", ( false & true ),
26          "true & false", ( true & false ),
27          "true & true", ( true & true ) );
28
29       // crea tabla de verdad para el operador | (OR inclusivo lógico booleano)
30       System.out.printf( "%s\n%s: %b\n%s: %b\n%s: %b\n%s: %b\n\n",
31          "OR inclusivo logico booleano (|)",
32          "false | false", ( false | false ),
33          "false | true", ( false | true ),
34          "true | false", ( true | false ),
35          "true | true", ( true | true ) );
36
37       // crea tabla de verdad para el operador ^ (OR exclusivo lógico booleano)
38       System.out.printf( "%s\n%s: %b\n%s: %b\n%s: %b\n%s: %b\n\n",
39          "OR exclusivo logico booleano (^)",
40          "false ^ false", ( false ^ false ),
41          "false ^ true", ( false ^ true ),
42          "true ^ false", ( true ^ false ),
43          "true ^ true", ( true ^ true ) );
44
45       // crea tabla de verdad para el operador ! (negación lógica)
46       System.out.printf( "%s\n%s: %b\n%s: %b\n", "NOT logico (!)",
47          "!false"( !false ), "!true", ( !true ) );
```

Figura 5.18 | Los operadores lógicos. (Parte 1 de 2).

```
    } // fin de main
} // fin de la clase OperadoresLogicos
```

```
AND condicional (&&)
false && false: false
false && true: false
true && false: false
true && true: true

OR condicional (||)
false || false: false
false || true: true
true || false: true
true || true: true

AND logico booleano (&)
false & false: false
false & true: false
true & false: false
true & true: true

OR inclusivo logico booleano (|)
false | false: false
false | true: true
true | false: true
true | true: true

OR exclusivo logico booleano (^)
false ^ false: false
false ^ true: true
true ^ false: true
true ^ true: false

NOT logico (!)
!false: true
!true: false
```

Figura 5.18 | Los operadores lógicos. (Parte 2 de 2).

La figura 5.19 muestra la precedencia y la asociatividad de los operadores de Java presentados hasta ahora. Los operadores se muestran de arriba hacia abajo, en orden descendente de precedencia.

Operadores	Asociatividad	Tipo
++ --	derecha a izquierda	postfijo unario
++ - + - ! (*tipo*)	derecha a izquierda	prefijo unario
* / %	izquierda a derecha	multiplicativo
+ -	izquierda a derecha	aditivo
< <= > >=	izquierda a derecha	relacional
== !=	izquierda a derecha	igualdad
&	izquierda a derecha	AND lógico booleano

Figura 5.19 | Precedencia/asociatividad de los operadores descritos hasta ahora. (Parte 1 de 2).

Operadores	Asociatividad	Tipo
^	izquierda a derecha	OR exclusivo lógico booleano
|	izquierda a derecha	OR inclusivo lógico booleano
&&	izquierda a derecha	AND condicional
||	izquierda a derecha	OR condicional
?:	derecha a izquierda	condicional
= += -= *= /= %=	derecha a izquierda	asignación

Figura 5.19 | Precedencia/asociatividad de los operadores descritos hasta ahora. (Parte 2 de 2).

5.9 Resumen sobre programación estructurada

Así como los arquitectos diseñan edificios empleando la sabiduría colectiva de su profesión, de igual forma, los programadores diseñan programas. Nuestro campo es mucho más joven que la arquitectura, y nuestra sabiduría colectiva es mucho más escasa. Hemos aprendido que la programación estructurada produce programas que son más fáciles de entender, probar, depurar, modificar que los programas no estructurados, e incluso probar que son correctos en sentido matemático.

La figura 5.20 utiliza diagramas de actividad de UML para sintetizar las instrucciones de control de Java. Los estados inicial y final indican el único punto de entrada y el único punto de salida de cada instrucción de control. Si conectamos los símbolos individuales de un diagrama de actividad en forma arbitrara, existe la posibilidad de que se produzcan programas no estructurados. Por lo tanto, la profesión de la programación ha elegido un conjunto limitado de instrucciones de control que pueden combinarse sólo de dos formas simples, para crear programas estructurados.

Por cuestión de simpleza, sólo se utilizan instrucciones de control de una sola entrada/una sola salida; sólo hay una forma de entrar y una forma de salir de cada instrucción de control. Es sencillo conectar instrucciones de control en secuencia para formar programas estructurados. El estado final de una instrucción de control se conecta al estado inicial de la siguiente instrucción de control; es decir, las instrucciones de control se colocan una después de la otra en un programa en secuencia. A esto le llamamos "apilamiento de instrucciones de control". Las reglas para formar programas estructurados también permiten anidar las instrucciones de control.

La figura 5.21 muestra las reglas para formar programas estructurados. Las reglas suponen que pueden utilizarse estados de acción para indicar cualquier acción. Además, las reglas suponen que comenzamos con el diagrama de actividad más sencillo (figura 5.22), que consiste solamente de un estado inicial, un estado de acción, un estado final y flechas de transición.

Al aplicar las reglas de la figura 5.21, siempre se obtiene un diagrama de actividad estructurado apropiadamente, con una agradable apariencia de bloque de construcción. Por ejemplo, si se aplica la regla 2 repetidamente al diagrama de actividad más sencillo, se obtiene un diagrama de actividad que contiene muchos estados de acción en secuencia (figura 5.23). La regla 2 genera una pila de estructuras de control, por lo que llamaremos a la regla 2 **regla de apilamiento**. [*Nota:* las líneas punteadas verticales en la figura 5.23 no son parte de UML. Las utilizamos para separar los cuatro diagramas de actividad que demuestran cómo se aplica la regla 2 de la figura 5.21].

La regla 3 se conoce como **regla de anidamiento**. Al aplicar la regla 3 repetidamente al diagrama de actividad más sencillo, se obtiene un diagrama de actividad con instrucciones de control perfectamente anidadas. Por ejemplo, en la figura 5.24 el estado de acción en el diagrama de actividad más sencillo se reemplaza con una instrucción de selección doble (`if...else`). Luego la regla 3 se aplica otra vez a los estados de acción en la instrucción de selección doble, reemplazando cada uno de estos estados con una instrucción de selección doble. El símbolo punteado de estado de acción alrededor de cada una de las instrucciones de selección doble, representa el estado de acción que se reemplazó. [*Nota:* las flechas punteadas y los símbolos punteados de estado de acción que se muestran en la figura 5.24 no son parte de UML. Aquí se utilizan para ilustrar que cualquier estado de acción puede reemplazarse con un enunciado de control].

La regla 4 genera instrucciones más grandes, más implicadas y más profundamente anidadas. Los diagramas que surgen debido a la aplicación de las reglas de la figura 5.21 constituyen el conjunto de todos los posibles

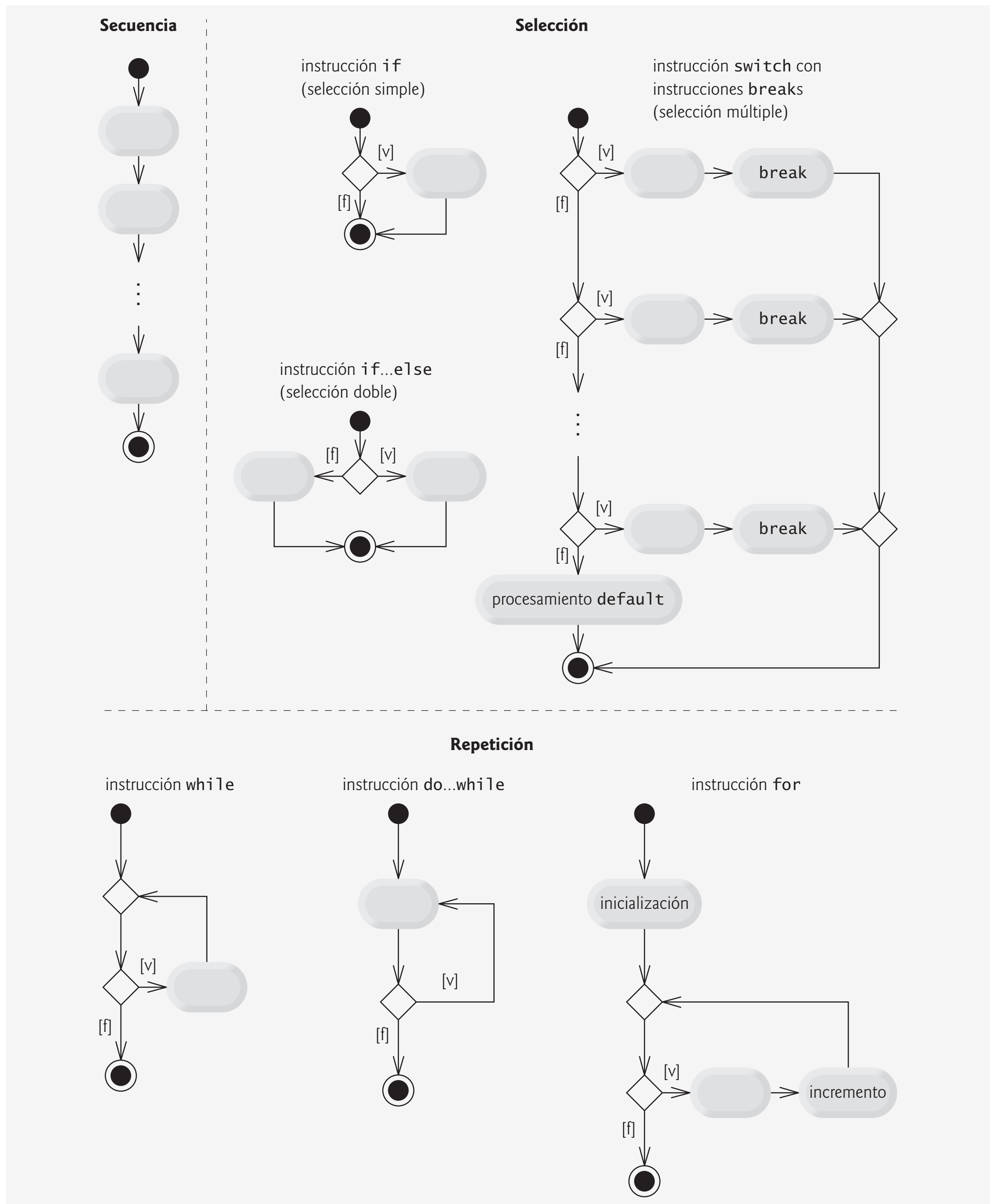


Figura 5.20 | Instrucciones de secuencia, selección y repetición de una sola entrada/una sola salida de Java.

diagramas de actividad estructurados y, por lo tanto, el conjunto de todos los posibles programas estructurados. La belleza de la metodología estructurada es que utilizamos sólo siete instrucciones de control simples de una sola entrada/una sola salida, y las ensamblamos en una de sólo dos formas simples.

Si se siguen las reglas de la figura 5.21, no podrá crearse un diagrama de actividad "sin estructura" (como el de la figura 5.25). Si usted no está seguro de que cierto diagrama sea estructurado, aplique las reglas de la

figura 5.21 en orden inverso para reducir el diagrama al diagrama de actividad más sencillo. Si puede reducirlo, entonces el diagrama original es estructurado; de lo contrario, no es estructurado.

Reglas para formar programas estructurados	
1)	Comenzar con el diagrama de actividad más sencillo (figura 5.22).
2)	Cualquier estado de acción puede reemplazarse por dos estados de acción en secuencia.
3)	Cualquier estado de acción puede reemplazarse por cualquier instrucción de control (secuencia de estados de acción, `if`, `if...else`, `switch`, `while`, `do...while` o `for`).
4)	Las reglas 2 y 3 pueden aplicarse tantas veces como se desee y en cualquier orden.

Figura 5.21 | Reglas para formar programas estructurados.



Figura 5.22 | El diagrama de actividad más sencillo.

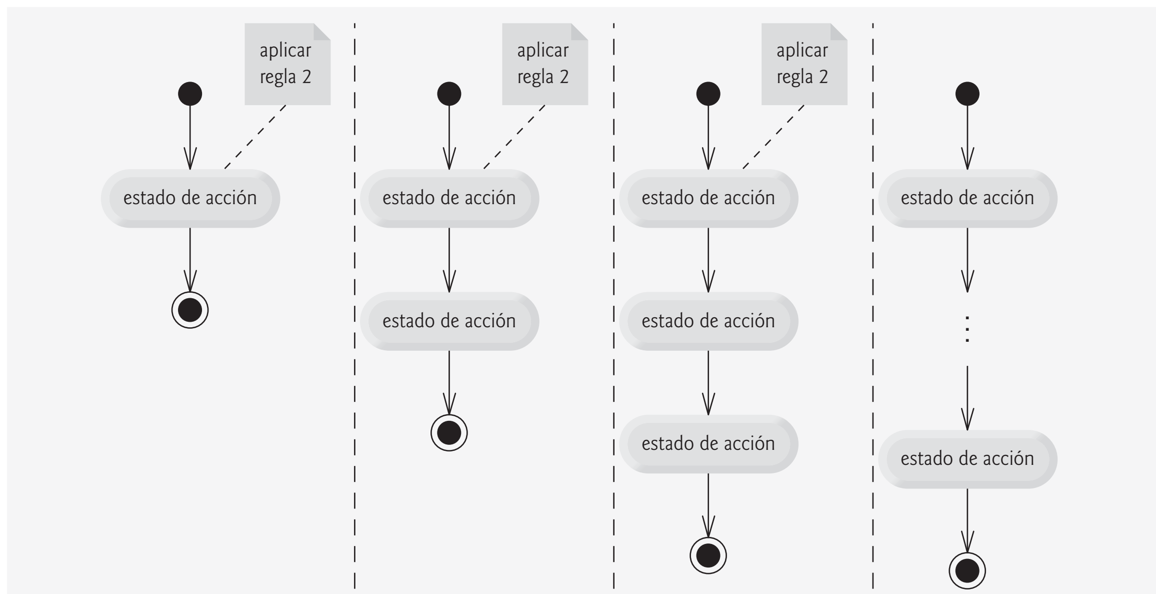


Figura 5.23 | El resultado de aplicar la regla de apilamiento (regla 2) de la figura 5.21 repetidamente al diagrama de actividad más sencillo.

Figura 5.24 | Aplicación de la regla de anidamiento (regla 3) de la figura 5.21 al diagrama de actividad más sencillo.

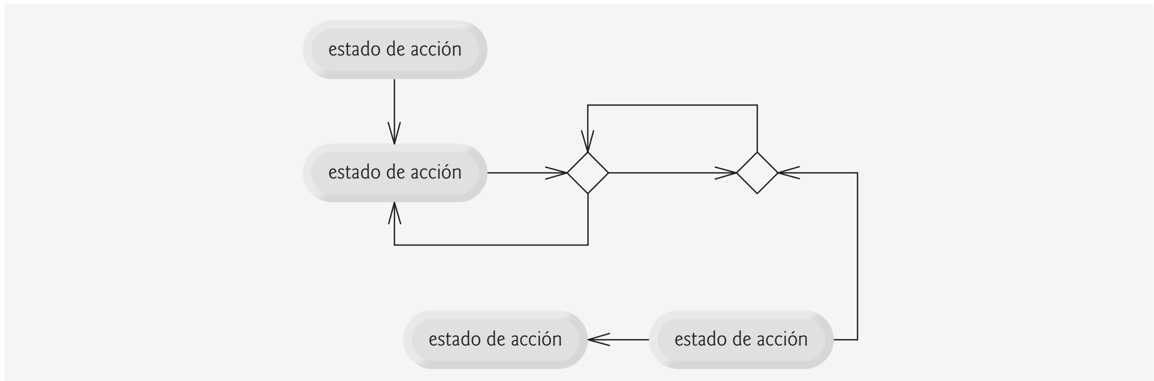


Figura 5.25 | Diagrama de actividad "sin estructura".

La programación estructurada promueve la simpleza. Bohm y Jacopini nos han dado el resultado de que sólo se necesitan tres formas de control para implementar un algoritmo:

- Secuencia.
- Selección.
- Repetición.

La estructura de secuencia es trivial. Simplemente enliste las instrucciones a ejecutar en el orden en el que deben ejecutarse. La selección se implementa en una de tres formas:

- Instrucción `if` (selección simple).
- Instrucción `if...else` (selección doble).
- Instrucción `switch` (selección múltiple).

De hecho, es sencillo demostrar que la instrucción `if` simple es suficiente para ofrecer cualquier forma de selección; todo lo que pueda hacerse con las instrucciones `if...else` y `switch` puede implementarse si se combinan instrucciones `if` (aunque tal vez no con tanta claridad y eficiencia).

La repetición se implementa en una de tres maneras:

- Instrucción `while`.
- Instrucción `do...while`.
- Instrucción `for`.

Es sencillo demostrar que la instrucción `while` es suficiente para proporcionar cualquier forma de repetición. Todo lo que puede hacerse con las instrucciones `do...while` y `for`, puede hacerse también con la instrucción `while` (aunque tal vez no sea tan sencillo).

Si se combinan estos resultados, se demuestra que cualquier forma de control necesaria en un programa en Java puede expresarse en términos de:

- Secuencia.
- Instrucción `if` (selección).
- Instrucción `while` (repetición).

y que estos tres elementos pueden combinarse en sólo dos formas: apilamiento y anidamiento. Evidentemente, la programación estructurada es la esencia de la simpleza.

5.10 (Opcional) Ejemplo práctico de GUI y gráficos: dibujo de rectángulos y óvalos

Esta sección demuestra cómo dibujar rectángulos y óvalos, usando los métodos `drawRect` y `drawOval` de `Graphics`, respectivamente. Estos métodos se demuestran en la figura 5.26.

La línea 6 empieza la declaración de la clase para `Figuras`, que extiende a `JPanel`. La variable de instancia `opcion`, declarada en la línea 8, determina si `paintComponent` debe dibujar rectángulos u óvalos. El constructor de `Figuras` en las líneas 11 a 14 inicializa `opcion` con el valor que se pasa en el parámetro `opcionUsuario`.

El método `paintComponent` (líneas 17 a 36) realiza el dibujo actual. Recuerde que la primera instrucción en todo método `paintComponent` debe ser una llamada a `super.paintComponent`, como en la línea 19. Las líneas 21 a 35 iteran 10 veces para dibujar 10 figuras. La instrucción `switch` (líneas 24 a 34) elije entre dibujar rectángulos y dibujar óvalos.

Si `opcion` es 1, entonces el programa dibuja un rectángulo. Las líneas 27 y 28 llaman al método `drawRect` de `Graphics`. El método `drawRect` requiere cuatro argumentos. Los primeros dos representan las coordenadas *x* y *y* de la esquina superior izquierda del rectángulo; los siguientes dos representan la anchura y la altura del rectángulo. En este ejemplo, empezamos en la posición 10 píxeles hacia abajo y 10 píxeles a la derecha de la esquina superior izquierda, y cada iteración del ciclo avanza la esquina superior izquierda otros 10 píxeles hacia abajo y a la derecha. La anchura y la altura del rectángulo empiezan en 50 píxeles, y se incrementan por 10 píxeles en cada iteración.

Si `opcion` es 2, el programa dibuja un óvalo. Al dibujar un óvalo se crea un rectángulo imaginario llamado **rectángulo delimitador**, y dentro de éste se crea un óvalo que toca los puntos medios de todos los cuatro lados

```
1  // Fig. 5.26: Figuras.java
2  // Demuestra cómo dibujar distintas figuras.
3  import java.awt.Graphics;
4  import javax.swing.JPanel;
5
6  public class Figuras extends JPanel
7  {
8     private int opcion; // opción del usuario acerca de cuál figura dibujar
9
10    // el constructor establece la opción del usuario
11    public Figuras( int opcionUsuario )
12    {
13       opcion = opcionUsuario;
14    } // fin del constructor de Figuras
15
16    // dibuja una cascada de figuras, empezando desde la esquina superior izquierda
17    public void paintComponent( Graphics g )
18    {
19       super.paintComponent( g );
20
21       for ( int i = 0; i < 10; i++ )
22       {
23          // elije la figura con base en la opción del usuario
24          switch ( opcion )
25          {
26             case 1: // dibuja rectángulos
27                g.drawRect( 10 + i * 10, 10 + i * 10,
28                   50 + i * 10, 50 + i * 10 );
29                break;
30             case 2: // dibuja óvalos
31                g.drawOval( 10 + i * 10, 10 + i * 10,
32                   50 + i * 10, 50 + i * 10 );
33                break;
34          } // fin del switch
35       } // fin del for
36    } // fin del método paintComponent
37 } // fin de la clase Figuras
```

Figura 5.26 | Cómo dibujar una cascada de figuras, con base en la opción elegida por el usuario.

del rectángulo delimitador. El método `drawOval` (líneas 31 y 32) requiere los mismos cuatro argumentos que el método `drawRect`. Los argumentos especifican la posición y el tamaño del rectángulo delimitador para el óvalo. Los valores que se pasan a `drawOval` en este ejemplo son exactamente los mismos valores que se pasan a `drawRect` en las líneas 27 y 28. Como la anchura y la altura del rectángulo delimitador son idénticas en este ejemplo, las líneas 27 y 28 dibujan un círculo. Puede modificar el programa para dibujar rectángulos y óvalos, para ver cómo se relacionan `drawOval` y `drawRect`.

La figura 5.27 es responsable de manejar la entrada del usuario y crear una ventana para mostrar el dibujo apropiado, con base en la respuesta del usuario. La línea 3 importa a `JFrame` para manejar la pantalla, y la línea 4 importa a `JOptionPane` para manejar la entrada.

Las líneas 11 a 13 muestran un cuadro de diálogo al usuario y almacenan la respuesta de éste en la variable `entrada`. La línea 15 utiliza el método `parseInt` de `Integer` para convertir el objeto `String` introducido por el usuario en un `int`, y almacena el resultado en la variable `opcion`. En la línea 18 se crea una instancia de la clase `Figuras`, y se pasa la opción del usuario al constructor. Las líneas 20 a 25 realizan las operaciones estándar para crear y establecer una ventana: crear un marco, configurarlo para que la aplicación termine cuando se cierre, agregar el dibujo al marco, establecer su tamaño y hacerlo visible.

```
// Fig. 5.27: PruebaFiguras.java
// Aplicación de prueba que muestra la clase Figuras.
import javax.swing.JFrame;
import javax.swing.JOptionPane;

public class PruebaFiguras
{
   public static void main( String args[] )
   {
      // obtiene la opción del usuario
      String entrada = JOptionPane.showInputDialog(
         "Escriba 1 para dibujar rectangulos\n" +
         "Escriba 2 para dibujar ovalos" );

      int opcion = Integer.parseInt( entrada ); // convierte entrada en int

      // crea el panel con la entrada del usuario
      Figuras panel = new Figuras( opcion );

      JFrame aplicacion = new JFrame(); // crea un nuevo objeto JFrame

      aplicacion.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
      aplicacion.add( panel ); // agrega el panel al marco
      aplicacion.setSize( 300, 300 ); // establece el tamaño deseado
      aplicacion.setVisible( true ); // muestra el marco
   } // fin de main
} // fin de la clase PruebaFiguras
```

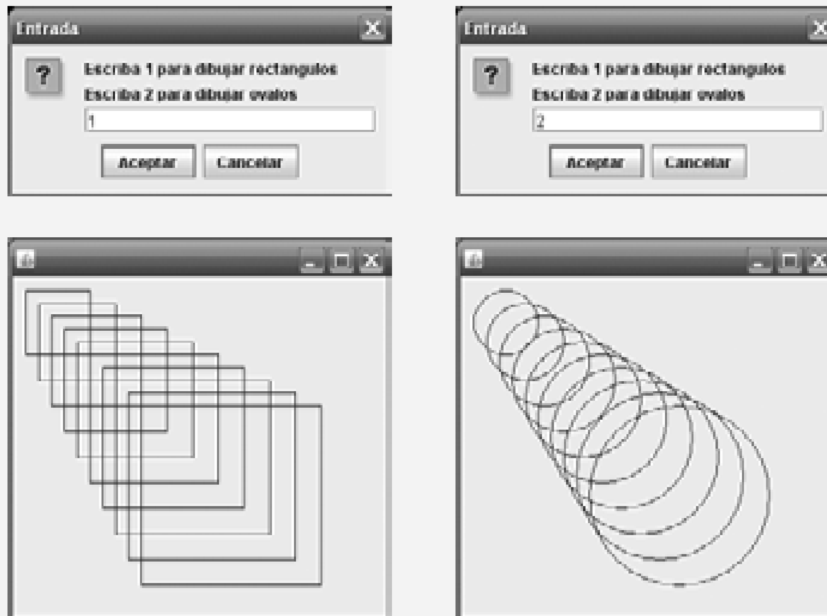


Figura 5.27 | Cómo obtener datos de entrada del usuario y crear un objeto `JFrame` para mostrar `figuras`.

Ejercicios del ejemplo práctico de GUI y gráficos

5.1 Dibuje 12 círculos concéntricos en el centro de un objeto `JPanel` (figura 5.28). El círculo más interno debe tener un radio de 10 píxeles, y cada círculo sucesivo debe tener un radio 10 píxeles mayor que el anterior. Empiece por buscar el centro del objeto `JPanel`. Para obtener la esquina superior izquierda de un círculo, avance un radio hacia arriba y un radio a la izquierda, partiendo del centro. La anchura y la altura del rectángulo delimitador es el diámetro del círculo (el doble del radio).

5.2 Modifique el ejercicio 5.16 de los ejercicios de fin de capítulo para leer la entrada usando cuadros de diálogo, y mostrar el gráfico de barras usando rectángulos de longitudes variables.

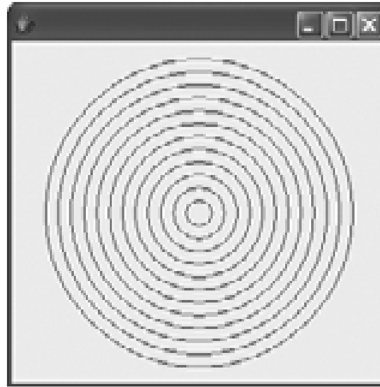


Figura 5.28 | Cómo dibujar círculos concéntricos.

5.11 (Opcional) Ejemplo práctico de Ingeniería de Software: cómo identificar los estados y actividades de los objetos

En la sección 4.15 identificamos muchos de los atributos de las clases necesarios para implementar el sistema ATM, y los agregamos al diagrama de clases de la figura 4.24. En esta sección le mostraremos la forma en que estos atributos representan el estado de un objeto. Identificaremos algunos estados clave que pueden ocupar nuestros objetos y hablaremos acerca de cómo cambian los objetos de estado, en respuesta a los diversos eventos que ocurren en el sistema. También hablaremos sobre el flujo de trabajo, o **actividades**, que realizan los objetos en el sistema ATM. En esta sección presentaremos las actividades de los objetos de transacción `SolicitudSaldo` y `Retiro`.

Diagramas de máquina de estado

Cada objeto en un sistema pasa a través de una serie de estados. El estado actual de un objeto se indica mediante los valores de los atributos del objeto en cualquier momento dado. Los **diagramas de máquina de estado** (que se conocen comúnmente como **diagramas de estado**) modelan varios estados de un objeto y muestran bajo qué circunstancias el objeto cambia de estado. A diferencia de los diagramas de clases que presentamos en las secciones anteriores del ejemplo práctico, que se enfocaban principalmente en la estructura del sistema, los diagramas de estado modelan parte del comportamiento del sistema.

La figura 5.29 es un diagrama de estado simple que modela algunos de los estados de un objeto de la clase `ATM`. UML representa a cada estado en un diagrama de estado como un **rectángulo redondeado** con el nombre del estado dentro de éste. Un **círculo relleno** con una punta de flecha designa el **estado inicial**. Recuerde que en el diagrama de clases de la figura 4.24 modelamos esta información de estado como el atributo `Boolean` de nombre `usuarioAutenticado`. Este atributo se inicializa en `false`, o en el estado "Usuario no autenticado", de acuerdo con el diagrama de estado.

Las flechas indican las **transiciones** entre los estados. Un objeto puede pasar de un estado a otro, en respuesta a los diversos eventos que ocurren en el sistema. El nombre o la descripción del evento que ocasiona una transición se escribe cerca de la línea que corresponde a esa transición. Por ejemplo, el objeto `ATM` cambia del estado "Usuario no autenticado" al estado "Usuario autenticado", una vez que la base de datos autentica al usuario. En el documento de requerimientos vimos que para autenticar a un usuario, la base de datos compara el número de

cuenta y el NIP introducidos por el usuario con los de la cuenta correspondiente en la base de datos. Si la base de datos indica que el usuario ha introducido un número de cuenta válido y el NIP correcto, el objeto `ATM` pasa al estado "Usuario autenticado" y cambia su atributo `usuarioAutenticado` al valor `true`. Cuando el usuario sale del sistema al seleccionar la opción "salir" del menú principal, el objeto `ATM` regresa al estado "Usuario no autenticado".

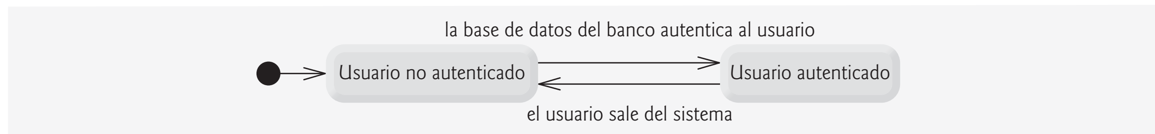


Figura 5.29 | Diagrama de estado para el objeto `ATM`.



Observación de ingeniería de software 5.5

Por lo general, los diseñadores de software no crean diagramas de estado que muestren todos los posibles estados y transiciones de estados para todos los atributos; simplemente hay demasiados. Lo común es que los diagramas de estado muestren sólo los estados y las transiciones de estado importantes.

Diagramas de actividad

Al igual que un diagrama de estado, un diagrama de actividad modela los aspectos del comportamiento de un sistema. A diferencia de un diagrama de estado, un diagrama de actividad modela el **flujo de trabajo** (secuencia de objetos) de un objeto durante la ejecución de un programa. Un diagrama de actividad modela las **acciones** a realizar y en qué orden las realizará el objeto. El diagrama de actividad de la figura 5.30 modela las acciones involucradas en la ejecución de una transacción de solicitud de saldo. Asumimos que ya se ha inicializado un objeto `SolicitudSaldo` y que ya se le ha asignado un número de cuenta válido (el del usuario actual), por lo que el objeto sabe qué saldo extraer de la base de datos. El diagrama incluye las acciones que ocurren después de que el usuario selecciona la opción de solicitud de saldo del menú principal y antes de que el ATM devuelva al usuario al menú principal; un objeto `SolicitudSaldo` no realiza ni inicia estas acciones, por lo que no las modelamos aquí. El diagrama empieza extrayendo de la base de datos el saldo de la cuenta. Después, `SolicitudSaldo` muestra el saldo en la pantalla. Esta acción completa la ejecución de la transacción. Recuerde que hemos optado por representar el saldo de una cuenta como los atributos `saldoDisponible` y `saldoTotal` de la clase `Cuenta`, por lo que las acciones que se modelan en la figura 5.30 hacen referencia a la obtención y visualización de ambos atributos del saldo.

UML representa una acción en un diagrama de actividad como un estado de acción, el cual se modela mediante un rectángulo en el que sus lados izquierdo y derecho se sustituyen por arcos hacia fuera. Cada estado de acción contiene una expresión de acción; por ejemplo, "obtener de la base de datos el saldo de la cuenta"; eso especifica una acción a realizar. Una flecha conecta dos estados de acción, con lo cual indica el orden en el que

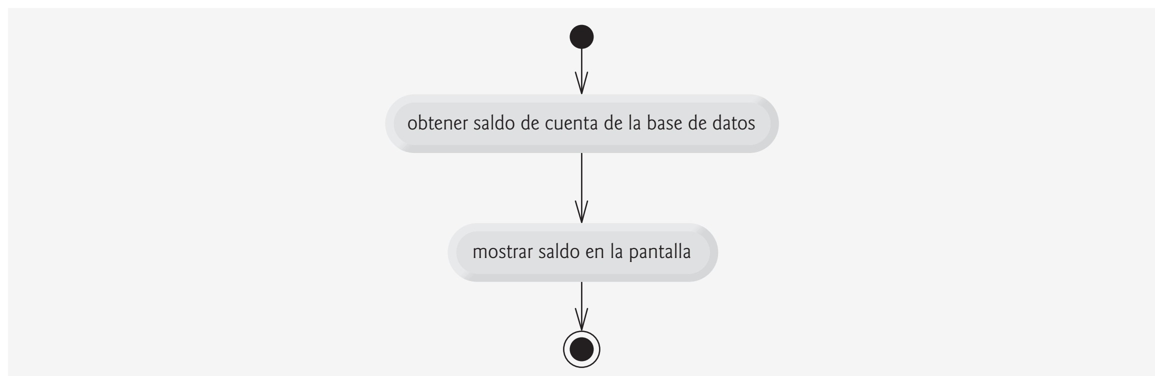


Figura 5.30 | Diagrama de actividad para un objeto `SolicitudSaldo`.

ocurren las acciones representadas por los estados de acción. El círculo relleno (en la parte superior de la figura 5.30) representa el estado inicial de la actividad: el inicio del flujo de trabajo antes de que el objeto realice las acciones modeladas. En este caso, la transacción primero ejecuta la expresión de acción "obtener de la base de datos el saldo de la cuenta". Después, la transacción muestra ambos saldos en la pantalla. El círculo relleno encerrado en un círculo sin relleno (en la parte inferior de la figura 5.30) representa el estado final: el fin del flujo de trabajo una vez que el objeto realiza las acciones modeladas. Utilizamos diagramas de actividad de UML para ilustrar el flujo de control para las instrucciones de control que presentamos en los capítulos 4 y 5.

La figura 5.31 muestra un diagrama de actividad para una transacción de retiro. Asumimos que ya se ha asignado un número de cuenta válido a un objeto `Retiro`. No modelaremos al usuario seleccionando la opción

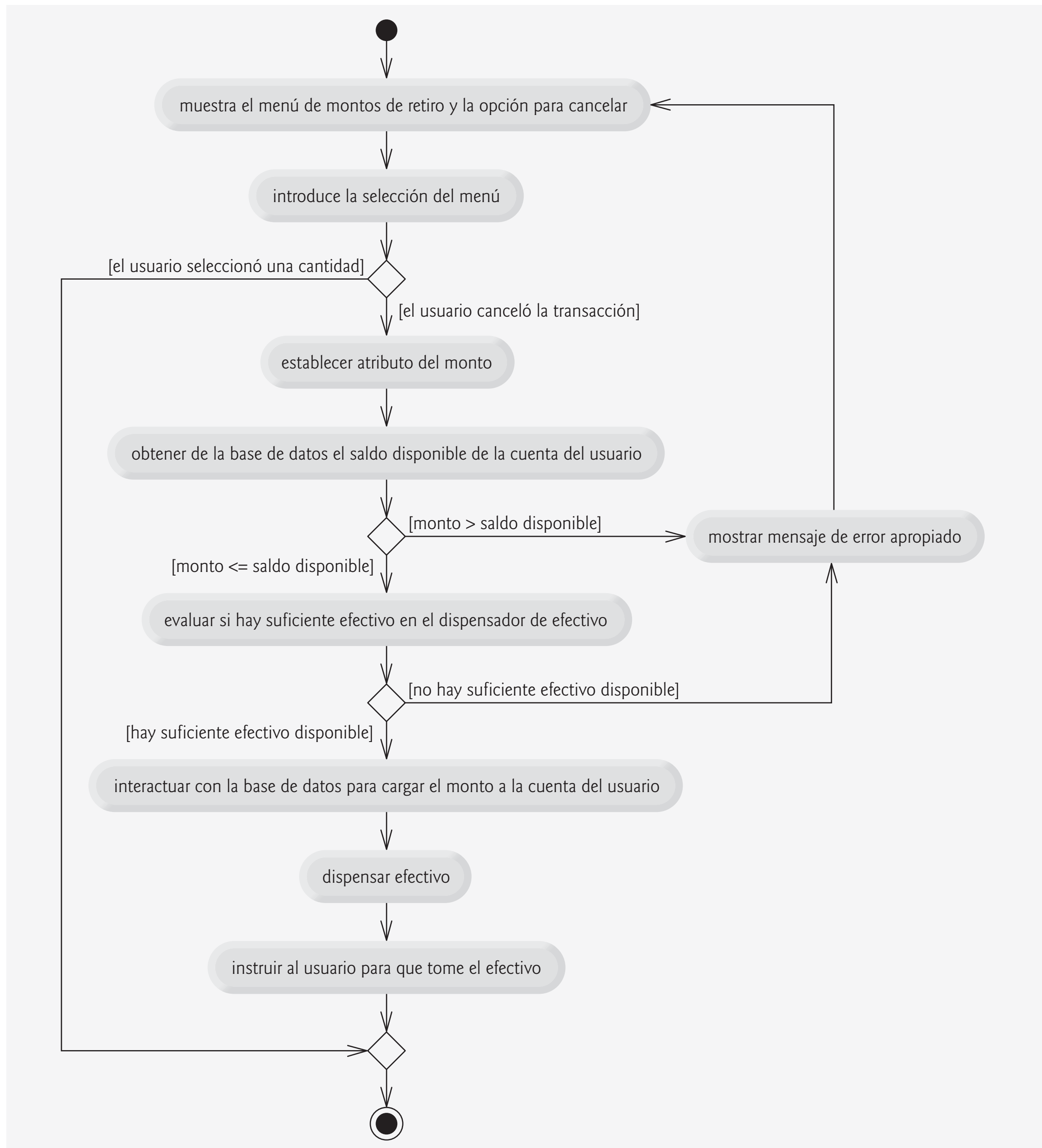


Figura 5.31 | Diagrama de actividad para una transacción de retiro.

de retiro del menú principal ni al ATM devolviendo al usuario al menú principal, ya que estas acciones no las realiza un objeto `Retiro`. La transacción primero muestra un menú de montos estándar de retiro (que se muestra en la figura 2.19) y una opción para cancelar la transacción. Después la transacción recibe una selección del menú de parte del usuario. Ahora el flujo de actividad llega a una decisión (una bifurcación indicada por el pequeño símbolo de rombo). [*Nota:* en versiones anteriores de UML, una decisión se conocía como una bifurcación]. Este punto determina la siguiente acción con base en la condición de guardia asociada (entre corchetes, enseguida de la transición), que indica que la transición ocurre si se cumple esta condición de guardia. Si el usuario cancela la transacción al elegir la opción "cancelar" del menú, el flujo de actividad salta inmediatamente al siguiente estado. Observe la fusión (indicada mediante el pequeño símbolo de rombo), en donde el flujo de actividad de cancelación se combina con el flujo de actividad principal, antes de llegar al estado final de la actividad. Si el usuario selecciona un monto de retiro del menú, `Retiro` establece `monto` (un atributo modelado originalmente en la figura 4.24) al valor elegido por el usuario.

Después de establecer el monto de retiro, la transacción obtiene el saldo disponible de la cuenta del usuario (es decir, el atributo `saldoDisponible` del objeto `Cuenta` del usuario) de la base de datos. Después el flujo de actividad llega a otra decisión. Si el monto de retiro solicitado excede al saldo disponible del usuario, el sistema muestra un mensaje de error apropiado, en el cual informa al usuario sobre el problema y después regresa al principio del diagrama de actividad, y pide al usuario que introduzca un nuevo monto. Si el monto de retiro solicitado es menor o igual al saldo disponible del usuario, la transacción continúa. A continuación, la transacción evalúa si el dispensador de efectivo tiene suficiente efectivo para satisfacer la solicitud de retiro. Si éste no es el caso, la transacción muestra un mensaje de error apropiado, después regresa al principio del diagrama de actividad y pide al usuario que seleccione un nuevo monto. Si hay suficiente efectivo disponible, la transacción interactúa con la base de datos para cargar el monto retirado de la cuenta del usuario (es decir, restar el monto tanto del atributo `saldoDisponible` como del atributo `saldoTotal` del objeto `Cuenta` del usuario). Después la transacción entrega el monto deseado de efectivo e instruye al usuario para que lo tome. Por último, el flujo de actividad se fusiona con el flujo de actividad de cancelación antes de llegar al estado final.

Hemos llevado a cabo los primeros pasos para modelar el comportamiento del sistema ATM y hemos mostrado cómo participan los atributos de un objeto para realizar las actividades del mismo. En la sección 6.14 investigaremos los comportamientos para todas las clases, de manera que obtengamos una interpretación más precisa del comportamiento del sistema, al "completar" los terceros compartimientos de las clases en nuestro diagrama de clases.

Ejercicios de autoevaluación del Ejemplo práctico de Ingeniería de Software

5.1 Indique si el siguiente enunciado es *verdadero* o *falso* y, si es *falso*, explique por qué: los diagramas de estado modelan los aspectos estructurales de un sistema.

5.2 Un diagrama de actividad modela las (los) __________ que realiza un objeto y el orden en el que las(los) realiza.
a) acciones
b) atributos
c) estados
d) transiciones de estado

5.3 Con base en el documento de requerimientos, cree un diagrama de actividad para una transacción de depósito.

Respuestas a los ejercicios de autoevaluación del Ejemplo práctico de Ingeniería de Software

5.1 Falso. Los diagramas de estado modelan parte del comportamiento del sistema.

5.2 a.

5.3 La figura 5.32 presenta un diagrama de actividad para una transacción de depósito. El diagrama modela las acciones que ocurren una vez que el usuario selecciona la opción de depósito del menú principal, y antes de que el ATM regrese al usuario al menú principal. Recuerde que una parte del proceso de recibir un monto de depósito de parte del usuario implica convertir un número entero de centavos a una cantidad en dólares. Recuerde también que para acreditar un monto de depósito a una cuenta sólo hay que incrementar el atributo `saldoTotal` del objeto `Cuenta` del usuario. El banco actualiza el atributo `saldoDisponible` del objeto `Cuenta` del usuario sólo después de confirmar el monto de efectivo en el sobre de depósito y después de verificar los cheques que haya incluido; esto ocurre en forma independiente del sistema ATM.

5.12 Conclusión

En este capítulo completamos nuestra introducción a las instrucciones de control de Java, las cuales nos permiten controlar el flujo de la ejecución en los métodos. El capítulo 4 trató acerca de las instrucciones de control `if`,

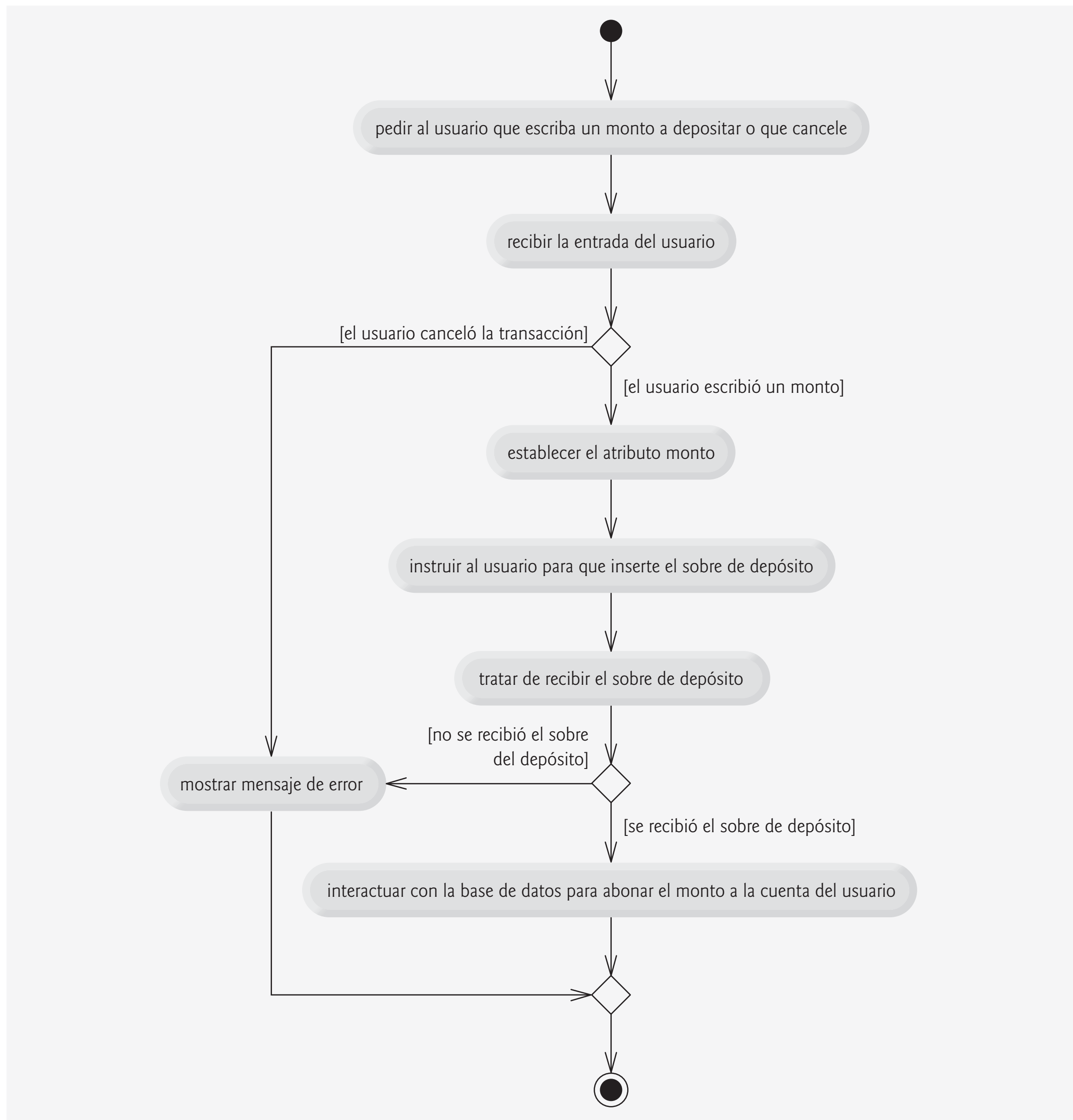


Figura 5.32 | Diagrama de actividad para una transacción de depósito.

`if...else` y `while` de Java. En este capítulo vimos el resto de las instrucciones de control de Java: `for`, `do...while` y `switch`. Aquí le mostramos que cualquier algoritmo puede desarrollarse mediante el uso de combinaciones de instrucciones de secuencia (es decir, instrucciones que se listan en el orden en el que deben ejecutarse), los tres tipos de instrucciones de selección (`if`, `if...else` y `switch`) y los tres tipos de instrucciones de repetición (`while`, `do...while` y `for`). En este capítulo y en el anterior hablamos acerca de cómo puede combinar estos bloques de construcción para utilizar las técnicas, ya probadas, de construcción de programas y solución de problemas. En este capítulo también se introdujeron los operadores lógicos de Java, que nos permiten utilizar expresiones condicionales más complejas en las instrucciones de control.

En el capítulo 3 presentamos los conceptos básicos de los objetos, las clases y los métodos. En los capítulos 4 y 5 se introdujeron los tipos de instrucciones de control que podemos utilizar para especificar la lógica de los programas en métodos. En el capítulo 6 examinaremos los métodos con más detalle.

Resumen

Sección 5.2 Fundamentos de la repetición controlada por contador

- La repetición controlada por contador requiere una variable de control (o contador de ciclo), el valor inicial de la variable de control, el incremento (o decremento) en base al cual se modifica la variable de control cada vez que pasa por el ciclo (lo que también se conoce como cada iteración del ciclo) y la condición de continuación de ciclo, que determina si el ciclo debe seguir ejecutándose.
- Podemos declarar e inicializar una variable en la misma instrucción.

Sección 5.3 Instrucción de repetición `for`

- La instrucción `while` puede usarse para implementar cualquier ciclo controlado por contador.
- La instrucción de repetición `for` especifica los detalles acerca de la repetición controlada por contador, en una sola línea de código.
- Cuando la instrucción `for` comienza a ejecutarse, su variable de control se declara y se inicializa. Después, el programa verifica la condición de continuación de ciclo. Si al principio la condición es verdadera, el cuerpo se ejecuta. Después de ejecutar el cuerpo del ciclo, se ejecuta la expresión de incremento. Después, se lleva a cabo otra vez la prueba de continuación de ciclo, para determinar si el programa debe continuar con la siguiente iteración del ciclo.
- El formato general de la instrucción `for` es

```
for ( inicialización; condiciónDeContinuacionDeCiclo; incremento )
    instrucción
```

  en donde la expresión *inicialización* asigna un nombre a la variable de control del ciclo y, de manera opcional, proporciona su valor inicial. *condiciónDeContinuaciónDeCiclo* es la condición que determina si el ciclo debe continuar su ejecución, e *incremento* modifica el valor de la variable de control (posiblemente un incremento o decremento), de manera que la condición de continuación de ciclo se vuelve falsa en un momento dado. Los dos signos de punto y coma en el encabezado `for` son obligatorios.
- En la mayoría de los casos, la instrucción `for` se puede representar con una instrucción `while` equivalente, de la siguiente forma:

```
inicialización;

while ( condiciónDeContinuaciónDeCiclo )
{
    instrucción
    incremento;
}
```

- Por lo general, las instrucciones `for` se utilizan para la repetición controlada por contador y las instrucciones `while` para la repetición controlada por centinela.
- Si la expresión de *inicialización* en el encabezado del `for` declara la variable de control, ésta sólo puede usarse en esa instrucción `for`; no existirá fuera de la instrucción `for`.
- Las tres expresiones en un encabezado `for` son opcionales. Si se omite la *condiciónDeContinuaciónDeCiclo*, Java asume que la condición de continuación de ciclo siempre es verdadera, con lo cual se crea un ciclo infinito. Podríamos omitir la expresión *inicialización* si el programa inicializa la variable de control antes del ciclo. Podríamos omitir la expresión *incremento* si el programa calcula el incremento con instrucciones en el cuerpo del ciclo, o si no se necesita un incremento.
- La expresión de incremento en un `for` actúa como si fuera una instrucción independiente al final del cuerpo del `for`.
- El incremento de una instrucción `for` puede ser también negativo, en cuyo caso es en realidad un decremento, y el ciclo cuenta en forma descendente.
- Si al principio la condición de continuación de ciclo es `false`, el programa no ejecuta el cuerpo de la instrucción `for`. En vez de ello, la ejecución continúa con la instrucción después del `for`.

Sección 5.4 Ejemplos sobre el uso de la instrucción `for`

- Java trata a las constantes de punto flotante, como `1000.0` y `0.05`, como de tipo `double`. De manera similar, Java trata a las constantes de números enteros, como `7` y `-22`, como de tipo `int`.

- El especificador de formato `%20s` indica que el objeto `String` de salida debe mostrarse con una anchura de campo de 20; es decir, `printf` muestra el valor con al menos 20 posiciones de caracteres. Si el valor a imprimir es menor de 20 posiciones de caracteres de ancho, se justifica a la derecha en el campo de manera predeterminada.
- `Math.pow(`*x, y*`)` calcula el valor de *x* elevado a la *y*-ésima potencia. El método recibe dos argumentos `double` y devuelve un valor `double`.
- La bandera de formato coma (,) en un especificador de formato (por ejemplo, `%,20.2f`) indica que un valor de punto flotante debe imprimirse con un separador de agrupamiento. El separador actual que se utiliza es específico de la configuración regional del usuario (es decir, el país). Por ejemplo, en los Estados Unidos el número se imprimirá usando comas para separar cada tres dígitos, y un punto decimal para separar la parte fraccionaria del número, como en 1,234.45.
- El `.2` en un especificador de formato (por ejemplo, `%,20.2f`) indica la precisión de un número con formato; en este caso, el número se redondea a la centésima más cercana y se imprime con dos dígitos a la derecha del punto decimal.

Sección 5.5 Instrucción de repetición `do...while`

- La instrucción de repetición `do...while` es similar a la instrucción `while`. En la instrucción `while`, el programa evalúa la condición de continuación de ciclo al principio del ciclo, antes de ejecutar su cuerpo; si la condición es falsa, el cuerpo nunca se ejecuta. La instrucción `do...while` evalúa la condición de continuación de ciclo *después* de ejecutar el cuerpo del ciclo; por lo tanto, el cuerpo siempre se ejecuta por lo menos una vez. Cuando termina una instrucción `do...while`, la ejecución continúa con la siguiente instrucción en secuencia.
- No es necesario usar llaves en la instrucción de repetición `do...while` si sólo hay una instrucción en el cuerpo. Sin embargo, la mayoría de los programadores incluyen las llaves, para evitar confusión entre las instrucciones `while` y `do...while`.

Sección 5.6 Instrucción de selección múltiple `switch`

- La instrucción `switch` de selección múltiple realiza distintas acciones, con base en los posibles valores de una variable o expresión entera. Cada acción se asocia con el valor de una expresión entera constante (es decir, un valor constante de tipo `byte`, `short`, `int` o `char`, pero no `long`) que la variable o expresión en la que se basa el `switch` puede asumir.
- El indicador de fin de archivo es una combinación de teclas dependiente del sistema, que el usuario escribe para indicar que no hay más datos qué introducir. En los sistemas UNIX/Linux/Mac OS X, el fin de archivo se introduce escribiendo la secuencia *<ctrl> d* en una línea por sí sola. Esta notación significa que hay que imprimir al mismo tiempo la tecla *ctrl* y la tecla *d*. En los sistemas Windows, el fin de archivo se puede introducir escribiendo *<ctrl> z*.
- El método `hasNext` de `Scanner` determina si hay más datos qué introducir. Este método devuelve el valor `boolean` `true` si hay más datos; en caso contrario, devuelve `false`. Mientras no se haya escrito el indicador de fin de archivo, el método `hasNext` devolverá `true`.
- La instrucción `switch` consiste en un bloque que contiene una secuencia de etiquetas `case` y un caso `default` opcional.
- Cuando el flujo de control llega al `switch`, el programa evalúa la expresión de control del `switch`. El programa compara el valor de la expresión de control (que debe evaluarse como un valor entero de tipo `byte`, `char`, `short` o `int`) con cada etiqueta `case`. Si ocurre una coincidencia, el programa ejecuta las instrucciones para esa etiqueta `case`.
- Al enlistar etiquetas `case` en forma consecutiva, sin instrucciones entre ellas, permite que las etiquetas ejecuten el mismo conjunto de instrucciones.
- La instrucción `switch` no cuenta con un mecanismo para evaluar rangos de valores, por lo que todo valor que deba evaluarse tiene que enlistarse en una etiqueta `case` separada.
- Cada `case` puede tener varias instrucciones. La instrucción `switch` se diferencia de las otras instrucciones de control, en cuanto a que no requiere llaves alrededor de varias instrucciones en una etiqueta `case`.
- Sin las instrucciones `break`, cada vez que ocurre una coincidencia en el `switch`, las instrucciones para ese `case` y los `case` subsiguientes se ejecutarán hasta llegar a una instrucción `break` o al final de la instrucción `switch`. A menudo esto se conoce como "pasar" a las instrucciones en las etiquetas `case` subsiguientes.
- Si no ocurre una coincidencia entre el valor de la expresión de control y una etiqueta `case`, se ejecuta el caso `default` opcional. Si no ocurre una coincidencia y la instrucción `switch` no tiene un caso `default`, el control del programa simplemente continúa con la primera instrucción después del `switch`.
- La instrucción `break` no se requiere para la última etiqueta `case` de la instrucción `switch` (ni para el caso `default` opcional, cuando aparece al último), ya que la ejecución continúa con la siguiente instrucción después del `switch`.

Sección 5.7 Instrucciones `break` *y* `continue`

- Además de las instrucciones de selección y repetición, Java cuenta con las instrucciones `break` y `continue` (que presentamos en esta sección y en el apéndice N, Instrucciones `break` y `continue` etiquetadas) para alterar el flujo

de control. La sección anterior mostró cómo se puede utilizar `break` para terminar la ejecución de una instrucción `switch`. Esta sección habla acerca de cómo utilizar `break` en instrucciones de repetición.
- Cuando la instrucción `break` se ejecuta en una instrucción `while`, `for`, `do...while` o `switch`, provoca la salida inmediata de esa instrucción. La ejecución continúa con la primera instrucción después de la instrucción de control.
- Cuando la instrucción `continue` se ejecuta en una instrucción `while`, `for` o `do...while`, omite el resto de las instrucciones en el cuerpo del ciclo y continúa con la siguiente iteración del mismo. En las instrucciones `while` y `do...while`, el programa evalúa la prueba de continuación de ciclo inmediatamente después de que se ejecuta la instrucción `continue`. En una instrucción `for`, se ejecuta la expresión de incremento y después el programa evalúa la prueba de continuación de ciclo.

Sección 5.8 Operadores lógicos
- Las condiciones simples se expresan en términos de los operadores relacionales `>`, `<`, `>=` y `<=`, y los operadores de igualdad `==` y `!=`, y cada expresión sólo evalúa una condición.
- Los operadores lógicos nos permiten formar condiciones más complejas, mediante la combinación de condiciones simples. Los operadores lógicos son `&&` (AND condicional), `||` (OR condicional), `&` (AND lógico booleano), `|` (OR inclusivo lógico booleano), `^` (OR exclusivo lógico booleano) y `!` (NOT lógico).
- Para asegurar que dos condiciones sean *ambas* verdaderas antes de elegir cierta ruta de ejecución, utilice el operador `&&` (AND condicional). Este operador da como resultado verdadero sí, y sólo si ambas de sus condiciones simples son verdaderas. Si una o ambas condiciones simples son falsas, la expresión completa es falsa.
- Para asegurar que una *o* ambas condiciones sean verdaderas antes de elegir cierta ruta de ejecución, utilice el operador `||` (OR condicional), que se evalúa como verdadero si una o ambas de sus condiciones simples son verdaderas.
- Las partes de una expresión que contienen operadores `&&` o `||` se evalúan sólo hasta que se conoce si la condición es verdadera o falsa. Esta característica de las expresiones AND condicional y OR condicional se conoce como evaluación de corto circuito.
- Los operadores AND lógico booleano (`&`) y OR inclusivo lógico booleano (`|`) funcionan de manera idéntica a los operadores `&&` (AND condicional) y `||` (OR condicional), con una excepción: los operadores lógicos boleanos siempre evalúan ambos operandos (es decir, no realizan una evaluación de corto circuito).
- Una condición simple que contiene el operador OR exclusivo lógico booleano (`^`) es `true` *si, y sólo si uno de sus operandos es* `true` *y el otro es* `false`. Si ambos operandos son `true` o ambos son `false`, toda la condición es `false`. También se garantiza que este operador evaluará ambos operandos.
- El operador `!` (NOT lógico, también conocido como negación lógica o complemento lógico) "invierte" el significado de una condición. El operador de negación lógico es un operador unario, que sólo tiene una condición como operando. Este operador se coloca antes de una condición, para elegir una ruta de ejecución si la condición original (sin el operador de negación lógico) es `false`.
- En la mayoría de los casos, podemos evitar el uso de la negación lógica si expresamos la condición de manera distinta, con un operador relacional o de igualdad apropiado.

Sección 5.10 (Opcional) Ejemplo práctico de GUI y gráficos: dibujo de rectángulos y óvalos
- Los métodos `drawRect` y `drawOval` de `Graphics` dibujan rectángulos y óvalos, respectivamente.
- El método `drawRect` de `Graphics` requiere cuatro argumentos. Los primeros dos representan las coordenadas *x* y *y* de la esquina superior izquierda del rectángulo; los otros dos representan la anchura y la altura del rectángulo.
- Al dibujar un óvalo, se crea un rectángulo imaginario llamado rectángulo delimitador, y se coloca en su interior un óvalo que toca los puntos medios de los cuatro lados del rectángulo delimitador. El método `drawOval` requiere los mismos cuatro argumentos que el método `drawRect`. Los argumentos especifican la posición y el tamaño del rectángulo delimitador para el óvalo.

Terminología

`-` (menos), bandera de formato
`!`, operador NOT lógico
`%b`, especificador de formato
`&`, operador AND lógico booleano
`&&`, operador AND condicional
`,` (coma), bandera de formato
`^`, operador OR exclusivo lógico booleano
`|`, operador OR lógico booleano
`||`, operador OR condicional
alcance de una variable
anchura de campo
AND condicional (`&&`)
AND lógico booleano (`&`)
`break`, instrucción
`case`, etiqueta
complemento lógico (`!`)
condición de continuación de ciclo
condición simple

constante de caracteres
`continue`, instrucción
decrementar una variable de control
`default`, caso en una instrucción `switch`
`do…while`, instrucción de repetición
`drawOval`, método de la clase `Graphics` (GUI)
`drawRect`, método de la clase `Graphics` (GUI)
efecto secundario
error por desplazamiento en 1
evaluación de corto circuito
expresión de control de una instrucción `switch`
expresión entera constante
`final`, palabra clave
`for`, encabezado
`for`, encabezado de la instrucción
`for`, instrucción de repetición
`hasNext`, método de la clase `Scanner`
incrementar una variable de control
indicador de fin de archivo
instrucción de repetición
instrucciones de control anidadas
instrucciones de control apiladas
instrucciones de control de una sola entrada/una sola salida
iteración de un ciclo
justificar a la derecha
justificar a la izquierda
método ayudante
negación lógica (`!`)
operadores lógicos
OR condicional (`||`)
OR exclusivo lógico booleano (`^`)
OR inclusivo lógico booleano (`|`)
rectángulo delimitador de un óvalo (GUI)
regla de anidamiento
regla de apilamiento
selección múltiple
`static`, método
`switch`, instrucción de selección
tabla de verdad
valor inicial
variable constante
variable de control

Ejercicios de autoevaluación

5.1 Complete los siguientes enunciados:

a) Por lo general, las instrucciones ______________ se utilizan para la repetición controlada por contador y las instrucciones ______________ se utilizan para la repetición controlada por centinela.
b) La instrucción `do...while` evalúa la condición de continuación de ciclo ______________ ejecutar el cuerpo del ciclo; por lo tanto, el cuerpo siempre se ejecuta por lo menos una vez.
c) La instrucción ______________ selecciona una de varias acciones, con base en los posibles valores de una variable o expresión entera.
d) Cuando se ejecuta la instrucción ______________ en una instrucción de repetición, se omite el resto de las instrucciones en el cuerpo del ciclo y se continúa con la siguiente iteración del ciclo.
e) El operador ______________ se puede utilizar para asegurar que *ambas* condiciones sean verdaderas, antes de elegir cierta ruta de ejecución.
f) Si al principio, la condición de continuación de ciclo en un encabezado `for` es ______________, el programa no ejecuta el cuerpo de la instrucción `for`.
g) Los métodos que realizan tareas comunes y no requieren objetos se llaman métodos ______________.

5.2 Conteste con *verdadero* o *falso* a cada una de las siguientes proposiciones; en caso de ser *falso*, explique por qué.

a) El caso `default` es requerido en la instrucción de selección `switch`.
b) La instrucción `break` es requerida en el último caso de una instrucción de selección `switch`.
c) La expresión `( ( x > y ) && ( a < b ) )` es verdadera si `x > y` es verdadera, o si `a < b` es verdadera.
d) Una expresión que contiene el operador `||` es verdadera si uno o ambos de sus operandos son verdaderos.
e) La bandera de formato coma (`,`) en un especificador de formato (por ejemplo, `%,20.2f`) indica que un valor debe imprimirse con un separador de miles.
f) Para evaluar un rango de valores en una instrucción `switch`, use un guión corto (–) entre los valores inicial y final del rango en una etiqueta `case`.
g) Al enlistar las instrucciones `case` en forma consecutiva, sin instrucciones entre ellas, pueden ejecutar el mismo conjunto de instrucciones.

5.3 Escriba una instrucción o un conjunto de instrucciones en Java, para realizar cada una de las siguientes tareas:

a) Sumar los enteros impares entre 1 y 99, utilizando una instrucción `for`. Suponga que se han declarado las variables enteras `suma` y `cuenta`.

b) Calcular el valor de `2.5` elevado a la potencia de `3`, utilizando el método `pow`.

c) Imprimir los enteros del 1 al 20, utilizando un ciclo `while` y la variable contador `i`. Suponga que la variable `i` se ha declarado, pero no se ha inicializado. Imprima solamente cinco enteros por línea. [*Sugerencia:* use el cálculo `i % 5`. Cuando el valor de esta expresión sea 0, imprima un carácter de nueva línea; de lo contrario, imprima un carácter de tabulación. Suponga que este código es una aplicación. Utilice el método `System.out.println()` para producir el carácter de nueva línea, y el método `System.out.print( '\t' )` para producir el carácter de tabulación].

d) Repita la parte (c), usando una instrucción `for`.

5.4 Encuentre el error en cada uno de los siguientes segmentos de código, y explique cómo corregirlo:

a)
```
i = 1;

while ( i <= 10 );
  i++;
}
```
b)
```
for ( k = 0.1; k != 1.0; k += 0.1 )
    System.out.println ( k );
```
c)
```
switch ( n )
{
    case 1:
       System.out.println( "El número es 1" );
    case 2:
       System.out.println( "El número es 2" );
       break;
    default:
       System.out.println( "El número no es 1 ni 2" );
       break;
}
```
d)
```
El siguiente código debe imprimir los valores 1 a 10:
n = 1;
while ( n < 10 )
    System.out.println( n++ );
```

Respuestas a los ejercicios de autoevaluación

5.1 a) `for`, `while`. b) después de. c) `switch`. d) `continue`. e) `&&` (AND condicional). f) `false`. g) `static`.

5.2 a) Falso. El caso `default` es opcional. Si no se necesita una acción predeterminada, entonces no hay necesidad de un caso `default`. b) Falso. La instrucción `break` se utiliza para salir de la instrucción `switch`. La instrucción `break` no se requiere para el último caso en una instrucción `switch`. c) Falso. Ambas expresiones relacionales deben ser verdaderas para que toda la expresión sea verdadera, cuando se utilice el operador `&&`. d) Verdadero. e) Verdadero. f) Falso. La instrucción `switch` no cuenta con un mecanismo para evaluar rangos de valores, por lo que todo valor que deba evaluarse se debe enlistar en una etiqueta `case` por separado. g) Verdadero.

5.3 a)
```
suma = 0;
for ( cuenta = 1; cuenta <= 99; cuenta += 2 )
suma += cuenta;
```
b)
```
double resultado = Math.pow( 2.5, 3 );
```
c)
```
i = 1;

while ( i <= 20 )
{
    System.out.print( i );

    if ( i % 5 == 0 )
       System.out.println()
    else
       System.out.print( '\t' );

    ++i;
}
```

d)
```
for ( i = 1; i <= 20; i++ )
{
    System.out.print( i );

    if ( i % 5 == 0 )
       System.out.println();
    else
       System.out.print( '\t' );
}
```

5.4 a) Error: el punto y coma después del encabezado `while` provoca un ciclo infinito, y falta una llave izquierda.
Corrección: reemplazar el punto y coma por una llave izquierda (`{`), o eliminar tanto el punto y coma (`;`) como la llave derecha (`}`).

b) Error: utilizar un número de punto flotante para controlar una instrucción `for` tal vez no funcione, ya que los números de punto flotante se representan sólo aproximadamente en la mayoría de las computadoras.
Corrección: utilice un entero, y realice el cálculo apropiado en orden para obtener los valores deseados:

```
for ( k = 1; k != 10, k++ )
  System.out.println( ( double ) k / 10 );
```

c) Error: el código que falta es la instrucción `break` en las instrucciones del primer `case`.
Corrección: agregue una instrucción `break` al final de las instrucciones para el primer `case`. Observe que esta omisión no es necesariamente un error, si el programador desea que la instrucción del `case 2:` se ejecute siempre que lo haga la instrucción del `case 1:`.

d) Error: se está utilizando un operador relacional inadecuado en la condición de continuación de la instrucción de repetición `while`.
Corrección: use `<=` en vez de `<`, o cambie `10` a `11`.

Ejercicios

5.5 Describa los cuatro elementos básicos de la repetición controlada por contador.

5.6 Compare y contraste las instrucciones de repetición `while` y `for`.

5.7 Hable sobre una situación en la que sería más apropiado usar una instrucción `do...while` que una instrucción `while`. Explique por qué.

5.8 Compare y contraste las instrucciones `break` y `continue`.

5.9 Encuentre y corrija el(los) error(es) en cada uno de los siguientes fragmentos de código:

a)
```
for ( i = 100, i >= 1, i++ )
    System.out.println ( i );
```

b) El siguiente código debe imprimirse sin importar si el valor entero es par o impar:

```
switch ( valor % 2 )
{
   case  0:
       System.out.println( "Entero par" );
   case 1:
       System.out.println( "Entero impar" );
}
```

c) El siguiente código debe imprimir los enteros impares del 19 al 1:

```
for ( i = 19; i >= 1; i += 2 )
  System.out.println( i );
```

d) El siguiente código debe imprimir los enteros pares del 2 al 100:

```
contador = 2;

do
{
  System.out.println( contador );
  contador += 2;
} While ( contador < 100 );
```

5.10 ¿Qué es lo que hace el siguiente programa?

```
public class Imprimir
{
   public static void main( String args[] )
   {
      for ( int i = 1; i <= 10; i++ )
      {
         for (int j = 1; j <= 5; j++ )
            System.out.print( '@' );

         System.out.println();
      } // fin del for exterior
   }  // fin de main
} // fin de la clase Imprimir.
```

5.11 Escriba una aplicación que encuentre el menor de varios enteros. Suponga que el primer valor leído especifica el número de valores que el usuario introducirá.

5.12 Escriba una aplicación que calcule el producto de los enteros impares del 1 al 15.

5.13 Los *factoriales* se utilizan frecuentemente en los problemas de probabilidad. El factorial de un entero positivo *n* (se escribe como *n!*) es igual al producto de los enteros positivos del 1 a *n*. Escriba una aplicación que evalúe los factoriales de los enteros del 1 al 5. Muestre los resultados en formato tabular. ¿Qué dificultad podría impedir que usted calculara el factorial de 20?

5.14 Modifique la aplicación de interés compuesto de la figura 5.6, repitiendo sus pasos para las tasas de interés del 5, 6, 7, 8, 9 y 10%. Use un ciclo `for` para variar la tasa de interés.

5.15 Escriba una aplicación que muestre los siguientes patrones por separado, uno debajo del otro. Use ciclos `for` para generar los patrones. Todos los asteriscos (*) deben imprimirse mediante una sola instrucción de la forma `System.out.print( '*' );` la cual hace que los asteriscos se impriman uno al lado del otro. Puede utilizarse una instrucción de la forma `System.out.println();` para posicionarse en la siguiente línea. Puede usarse una instrucción de la forma `System.out.print( ' ' );` para mostrar un espacio para los últimos dos patrones. No debe haber ninguna otra instrucción de salida en el programa. [*Sugerencia:* los últimos dos patrones requieren que cada línea empiece con un número apropiado de espacios en blanco].

```
(a)            (b)            (c)            (d)

*              **********     **********              *
**             *********       *********             **
***            ********         ********            ***
****           *******           *******           ****
*****          ******             ******          *****
******         *****               *****         ******
*******        ****                 ****        *******
********       ***                   ***       ********
*********      **                     **      *********
**********     *                       *     **********
```

5.16 Una aplicación interesante de las computadoras es dibujar gráficos convencionales y de barra. Escriba una aplicación que lea cinco números, cada uno entre 1 y 30. Por cada número leído, su programa debe mostrar ese número de asteriscos adyacentes. Por ejemplo, si su programa lee el número 7, debe mostrar *******.

5.17 Un almacén de pedidos por correo vende cinco productos cuyos precios de venta son los siguientes: producto 1, $2.98; producto 2, $4.50; producto 3, $9.98; producto 4, $4.49 y producto 5, $6.87. Escriba una aplicación que lea una serie de pares de números, como se muestra a continuación:

a) número del producto;
b) cantidad vendida.

Su programa debe utilizar una instrucción `switch` para determinar el precio de venta de cada producto. Debe calcular y mostrar el valor total de venta de todos los productos vendidos. Use un ciclo controlado por centinela para determinar cuándo debe el programa dejar de iterar para mostrar los resultados finales.

5.18 Modifique la aplicación de la figura 5.6, de manera que se utilicen sólo enteros para calcular el interés compuesto. [*Sugerencia:* trate todas las cantidades monetarias como números enteros de centavos. Luego divida el resultado en su porción de dólares y su porción de centavos, utilizando las operaciones de división y residuo, respectivamente. Inserte un punto entre las porciones de dólares y centavos].

5.19 Suponga que `i = 1, j = 2, k = 3` y `m = 2`. ¿Qué es lo que imprime cada una de las siguientes instrucciones?

a) `System.out.println( i == 1 );`
b) `System.out.println( j == 3 );`
c) `System.out.println( ( i >= 1 ) && ( j < 4 ) );`
d) `System.out.println( ( m <= 99 ) & ( k < m ) );`
e) `System.out.println( ( j >= i ) || ( k == m ) );`
f) `System.out.println( ( k + m < j ) | ( 3 - j >= k ) );`
g) `System.out.println( !( k > m ) );`

5.20 Calcule el valor de π a partir de la serie infinita

$$\pi = 4 - \frac{4}{3} + \frac{4}{5} - \frac{4}{7} + \frac{4}{9} - \frac{4}{11} + \cdots$$

Imprima una tabla que muestre el valor aproximado de π, calculando un término de esta serie, dos términos, tres, etcétera. ¿Cuántos términos de esta serie tiene que utilizar para obtener 3.14? ¿3.141? ¿3.1415? ¿3.14159?

5.21 (*Triples de Pitágoras*) Un triángulo recto puede tener lados cuyas longitudes sean valores enteros. El conjunto de tres valores enteros para las longitudes de los lados de un triángulo recto se conoce como triple de Pitágoras. Las longitudes de los tres lados deben satisfacer la relación que establece que la suma de los cuadrados de dos lados es igual al cuadrado de la hipotenusa. Escriba una aplicación para encontrar todos los triples de Pitágoras para `lado1`, `lado2`, y la `hipotenusa`, que no sean mayores de 500. Use un ciclo `for` triplemente anidado para probar todas las posibilidades. Este método es un ejemplo de la computación de "fuerza bruta". En cursos de ciencias computacionales más avanzados aprenderá que existen muchos problemas interesantes para los cuales no hay otra metodología algorítmica conocida, más que el uso de la fuerza bruta.

5.22 Modifique el ejercicio 5.15 para combinar su código de los cuatro triángulos separados de asteriscos, de manera que los cuatro patrones se impriman uno al lado del otro. [*Sugerencia:* utilice astutamente los ciclos `for` anidados].

5.23 (*Leyes de De Morgan*) En este capítulo, hemos hablado sobre los operadores lógicos `&&`, `&`, `||`, `|`, `^` y `!`. Algunas veces, las leyes de De Morgan pueden hacer que sea más conveniente para nosotros expresar una expresión lógica. Estas leyes establecen que la expresión `!` `(`*condición1* `&&` *condición2*`)` es lógicamente equivalente a la expresión `(!`*condición1* `||` `!`*condición2*`)`. También establecen que la expresión `!(`*condición1* `||` *condición2*`)` es lógicamente equivalente a la expresión `(!`*condición1* `&&` `!`*condición2*`)`. Use las leyes de De Morgan para escribir expresiones equivalentes para cada una de las siguientes expresiones, luego escriba una aplicación que demuestre que, tanto la expresión original como la nueva expresión, producen en cada caso el mismo valor:

a) `!( x < 5 ) && !( y >= 7 )`
b) `!( a == b ) || !( g != 5 )`
c) `!( ( x <= 8 ) && ( y > 4 ) )`
d) `!( ( i > 4 ) || ( j <= 6 ) )`

5.24 Escriba una aplicación que imprima la siguiente figura de rombo. Puede utilizar instrucciones de salida que impriman un solo asterisco (*), un solo espacio o un solo carácter de nueva línea. Maximice el uso de la repetición (con instrucciones `for` anidadas), y minimice el número de instrucciones de salida.

```
    *
   ***
  *****
 *******
*********
 *******
  *****
   ***
    *
```

5.25 Modifique la aplicación que escribió en el ejercicio 5.24, para que lea un número impar en el rango de 1 a 19, de manera que especifique el número de filas en el rombo. Su programa debe entonces mostrar un rombo del tamaño apropiado.

5.26 Una crítica de las instrucciones `break` y `continue` es que ninguna es estructurada. En realidad, estas instrucciones pueden reemplazarse en todo momento por instrucciones estructuradas, aunque hacerlo podría ser inadecuado. Describa, en general, cómo eliminaría las instrucciones `break` de un ciclo en un programa, para reemplazarlas con alguna de las instrucciones estructuradas equivalentes. [*Sugerencia:* la instrucción `break` se sale de un ciclo desde el cuerpo de éste. La otra forma de salir es que falle la prueba de continuación de ciclo. Considere utilizar en la prueba de continuación de ciclo una segunda prueba que indique una "salida anticipada debido a una condición de 'interrupción'"]. Use la técnica que desarrolló aquí para eliminar la instrucción `break` de la aplicación de la figura 5.12.

5.27 ¿Qué hace el siguiente segmento de programa?

```
for ( i = 1; i <= 5; i++ )
{
    for ( j = 1; j <= 3; j++ )
    {
        for ( k = 1; k <= 4; k++ )
          System.out.print( '*' );

        System.out.println();
    }  // fin del for interior

    System.out.println();
}  // fin del for exterior
```

5.28 Describa, en general, cómo eliminaría las instrucciones `continue` de un ciclo en un programa, para reemplazarlas con uno de sus equivalentes estructurados. Use la técnica que desarrolló aquí para eliminar la instrucción `continue` del programa de la figura 5.13.

5.29 *(Canción "Los Doce Días de Navidad")* Escriba una aplicación que utilice instrucciones de repetición y `switch` para imprimir la canción "Los Doce Días de Navidad". Una instrucción `switch` debe utilizarse para imprimir el día (es decir, "primer", "segundo", etcétera). Una instrucción `switch` separada debe utilizarse para imprimir el resto de cada verso. Visite el sitio Web `en.wikipedia.org/wiki/Twelvetide` para obtener la letra completa de la canción.

6

Métodos: un análisis más detallado



La más grande invención del siglo diecinueve fue la invención del método de la invención.

—Alfred North Whitehead

Llámame Ismael.

—Herman Melville

Cuando me llames así, sonríe.

—Owen Wister

Respóndeme en una palabra.

—William Shakespeare

¡Oh! volvió a llamar ayer, ofreciéndome volver.

—William Shakespeare

Hay un punto en el cual los métodos se devoran a sí mismos.

—Frantz Fanon

OBJETIVOS

En este capítulo aprenderá a:

- Conocer cómo se asocian los métodos y los campos `static` con toda una clase, en vez de asociarse con instancias específicas de la clase.
- Utilizar los métodos comunes de `Math` disponibles en la API de Java.
- Comprender los mecanismos para pasar información entre métodos.
- Comprender cómo se soporta el mecanismo de llamada/retorno de los métodos mediante la pila de llamadas a métodos y los registros de activación.
- Conocer cómo los paquetes agrupan las clases relacionadas.
- Utilizar la generación de números aleatorios para implementar aplicaciones para juegos.
- Comprender cómo se limita la visibilidad de las declaraciones a regiones específicas de los programas.
- Acerca de la sobrecarga de métodos y cómo crear métodos sobrecargados.

Plan general

6.1 Introducción

La mayoría de los programas de cómputo que resuelven los problemas reales son mucho más extensos que los programas que se presentan en los primeros capítulos de este libro. La experiencia ha demostrado que la mejor manera de desarrollar y mantener un programa extenso es construirlo a partir de pequeñas piezas sencillas, o **módulos**. A esta técnica se le llama **divide y vencerás**. En el capítulo 3 presentamos los métodos, y en éste lo estudiaremos con más detalle. Haremos énfasis en cómo declarar y utilizar métodos para facilitar el diseño, la implementación, operación y el mantenimiento de programas extensos.

En breve verá que es posible que ciertos métodos, conocidos como `static` (métodos estáticos), puedan llamarse sin necesidad de que exista un objeto de la clase a la que pertenecen. Aprenderá a declarar un método con más de un parámetro. También aprenderá acerca de cómo Java es capaz de llevar el rastro de qué método se ejecuta en un momento dado, cómo se mantienen las variables locales de los métodos en memoria y cómo sabe un método a dónde regresar una vez que termina su ejecución.

Hablaremos brevemente sobre las técnicas de simulación mediante la generación de números aleatorios y desarrollaremos una versión de un juego de dados conocido como "craps", el cual utiliza la mayoría de las técnicas de programación que usted ha aprendido hasta este capítulo. Además, aprenderá a declarar valores que no pueden cambiar (es decir, constantes) en sus programas.

Muchas de las clases que utilizará o creará mientras desarrolla aplicaciones tendrán más de un método con el mismo nombre. Esta técnica, conocida como sobrecarga, se utiliza para implementar métodos que realizan tareas similares, para argumentos de distintos tipos, o para un número distinto de argumentos.

En el capítulo 15, Recursividad, continuaremos nuestra discusión sobre los métodos. La recursividad proporciona una manera completamente distinta de pensar acerca de los métodos y los algoritmos.

6.2 Módulos de programas en Java

Existen tres tipos de módulos en Java: métodos, clases y paquetes. Para escribir programas en Java, se combinan los nuevos métodos y clases que usted escribe con los métodos y clases predefinidos, que están disponibles en la **Interfaz de Programación de Aplicaciones de Java** (también conocida como la **API de Java** o **biblioteca de**

clases de Java) y en diversas bibliotecas de clases. Por lo general, las clases relacionadas están agrupadas en paquetes, de manera que se pueden importar a los programas y reutilizarse. En el capítulo 8 aprenderá a agrupar sus propias clases en paquetes. La API de Java proporciona una vasta colección de clases que contienen métodos para realizar cálculos matemáticos, manipulaciones de cadenas, manipulaciones de caracteres, operaciones de entrada/salida, comprobación de errores y muchas otras operaciones útiles.



Buena práctica de programación 6.1

Procure familiarizarse con la vasta colección de clases y métodos que proporciona la API de Java (`java.sun.com/javase/6/docs/api/`). En la sección 6.8 presentaremos las generalidades acerca de varios paquetes comunes. En el apéndice J, le explicaremos cómo navegar por la documentación de la API de Java.



Observación de ingeniería de software 6.1

Evite reinventar la rueda. Cuando sea posible, reutilice las clases y métodos de la API de Java. Esto reduce el tiempo de desarrollo de los programas y evita que se introduzcan errores de programación.

Los métodos (también conocidos como **funciones** o **procedimientos** en otros lenguajes) permiten al programador dividir un programa en módulos, por medio de la separación de sus tareas en unidades autónomas. Usted ha declarado métodos en todos los programas que ha escrito; a estos métodos se les conoce algunas veces como **métodos declarados por el programador**. Las instrucciones en los cuerpos de los métodos se escriben sólo una vez, y se reutilizan tal vez desde varias ubicaciones en un programa; además, están ocultas de otros métodos.

Una razón para dividir un programa en módulos mediante los métodos es la metodología "divide y vencerás", que hace que el desarrollo de programas sea más fácil de administrar, ya que se pueden construir programas a partir de piezas pequeñas y simples. Otra razón es la **reutilización de software** (usar los métodos existentes como bloques de construcción para crear nuevos programas). A menudo se pueden crear programas a partir de métodos estandarizados, en vez de tener que crear código personalizado. Por ejemplo, en los programas anteriores no tuvimos que definir cómo leer los valores de datos del teclado; Java proporciona estas herramientas en la clase `Scanner`. Una tercera razón es para evitar la repetición de código. El proceso de dividir un programa en métodos significativos hace que el programa sea más fácil de depurar y mantener.



Observación de ingeniería de software 6.2

Para promover la reutilización de software, cada método debe limitarse de manera que realice una sola tarea bien definida, y su nombre debe expresar esa tarea con efectividad. Estos métodos hacen que los programas sean más fáciles de escribir, depurar, mantener y modificar.



Tip para prevenir errores 6.1

Un método pequeño que realiza una tarea es más fácil de probar y depurar que un método más grande que realiza muchas tareas.



Observación de ingeniería de software 6.3

Si no puede elegir un nombre conciso que exprese la tarea de un método, tal vez esté tratando de realizar diversas tareas en un mismo método. Por lo general, es mejor dividirlo en varias declaraciones de métodos más pequeños.

Un método se invoca mediante una llamada, y cuando el método que se llamó completa su tarea, devuelve un resultado, o simplemente el control al método que lo llamó. Una analogía a esta estructura de programa es la forma jerárquica de la administración (figura 6.1). Un jefe (el solicitante) pide a un trabajador (el método llamado) que realice una tarea y que le reporte (devuelva) los resultados después de completar la tarea. El método jefe, no sabe cómo el método trabajador, realiza sus tareas designadas. Tal vez el trabajador llame a otros métodos trabajadores, sin que lo sepa el jefe. Este "ocultamiento" de los detalles de implementación fomenta la buena ingeniería de software. La figura 6.1 muestra al método `jefe` comunicándose con varios métodos trabajadores en forma jerárquica. El método `jefe` divide las responsabilidades entre los diversos métodos trabajadores. Observe que `trabajador1` actúa como "método jefe" de `trabajador4` y `trabajador5`.

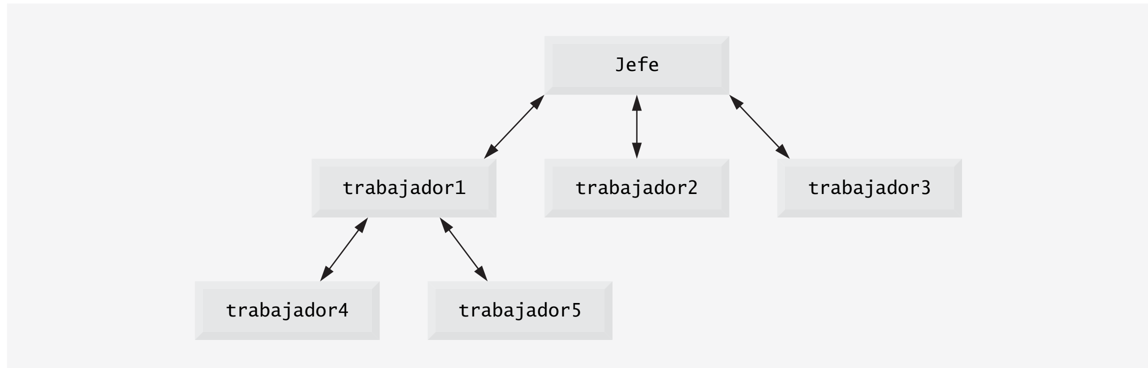


Figura 6.1 | Relación jerárquica entre el método jefe y los métodos trabajadores.

6.3 Métodos `static`, campos `static` y la clase `Math`

Toda clase proporciona métodos que realizan tareas comunes en objetos de esa clase. Por ejemplo, para introducir datos mediante el teclado, hemos llamado métodos en un objeto `Scanner` que se inicializó en su constructor para obtener la entrada del flujo de entrada estándar (`System.in`). Como aprenderá en el capítulo 14, Archivos y flujos, puede inicializar un objeto `Scanner` que reciba información del flujo de entrada estándar, y un segundo objeto `Scanner` que reciba información de un archivo. Cada método de entrada que se llame en el objeto `Scanner` del flujo de entrada estándar obtendría su entrada del teclado, y cada método de entrada que se llame en el objeto `Scanner` del archivo obtendría su entrada del archivo especificado en el disco.

Aunque la mayoría de los métodos se ejecutan en respuesta a las llamadas a métodos en objetos específicos, éste no es siempre el caso. Algunas veces un método realiza una tarea que no depende del contenido de ningún objeto. Dicho método se aplica a la clase en la que está declarado como un todo, y se conoce como método `static` o **método de clase**. Es común que las clases contengan métodos `static` convenientes para realizar tareas comunes. Por ejemplo, recuerde que en la figura 5.6 utilizamos el método `static pow` de la clase `Math` para elevar un valor a una potencia. Para declarar un método como `static`, coloque la palabra clave `static` antes del tipo de valor de retorno en la declaración del método. Puede llamar a cualquier método `static` especificando el nombre de la clase en la que está declarado el método, seguido de un punto (`.`) y del nombre del método, como sigue:

NombreClase.nombreMétodo (*argumentos*)

Aquí utilizaremos varios métodos de la clase `Math` para presentar el concepto de los métodos `static`. La clase `Math` cuenta con una colección de métodos que nos permiten realizar cálculos matemáticos comunes. Por ejemplo, podemos calcular la raíz cuadrada de `900.0` con una llamada al siguiente método `static`:

```
Math.sqrt( 900.0 )
```

La expresión anterior se evalúa como `30.0`. El método `sqrt` recibe un argumento de tipo `double` y devuelve un resultado del mismo tipo. Para imprimir el valor de la llamada anterior al método en una ventana de comandos, podríamos escribir la siguiente instrucción:

```
System.out.println( Math.sqrt( 900.0 ) );
```

En esta instrucción, el valor que devuelve `sqrt` se convierte en el argumento para el método `println`. Observe que no hubo necesidad de crear un objeto `Math` antes de llamar al método `sqrt`. Observe también que *todos* los métodos de la clase `Math` son `static`; por lo tanto, cada uno se llama anteponiendo al nombre del método el nombre de la clase `Math` y el separador punto (`.`).



Observación de ingeniería de software 6.4

La clase `Math` *es parte del paquete* `java.lang`*, que el compilador importa de manera implícita, por lo que no es necesario importarla para utilizar sus métodos.*

Los argumentos para los métodos pueden ser constantes, variables o expresiones. Si `c = 13.0`, `d = 3.0` y `f = 4.0`, entonces la instrucción

```
System.out.println( Math.sqrt( c + d * f ) );
```

calcula e imprime la raíz cuadrada de `13.0 + 3.0 * 4.0 = 25.0`; a saber, `5.0`. La figura 6.2 sintetiza varios de los métodos de la clase `Math`. En la figura 6.2, *x* y *y* son de tipo `double`.

Método	Descripción	Ejemplo

`abs( x )`	valor absoluto de *x*	`abs( 23.7 )` es `23.7`
		`abs ( 0.0 )` es `0.0`
		`abs( -23.7 )` es `23.7`

`ceil( x )`	redondea *x* al entero más pequeño que no sea menor de *x*	`ceil( 9.2 )` es `10.0` `ceil( -9.8 )` es `-9.0`
`cos( x )`	coseno trigonométrico de *x* (*x* está en radianes)	`cos( 0.0 )` es `1.0`
`exp( x )`	método exponencial e^x	`exp( 1.0 )` es `2.71828` `exp( 2.0 )` es `7.38906`
`floor( x )`	redondea *x* al entero más grande que no sea mayor de *x*	`floor( 9.2 )` es `9.0` `floor( -9.8 )` es `-10.0`
`log( x )`	logaritmo natural de *x* (base *e*)	`log( Math.E )` es `1.0` `log( Math.E * Math.E )` es `2.0`
`max( x, y )`	el valor más grande de *x* y *y*	`max( 2.3, 12.7 )` es `12.7` `max( -2.3, -12.7 )` es `-2.3`
`min( x, y )`	el valor más pequeño de *x* y *y*	`min( 2.3, 12.7 )` es `2.3` `min( -2.3, -12.7 )` es `-12.7`
`pow( x, y )`	*x* elevado a la potencia *y* (x^y)	`pow( 2.0, 7.0 )` es `128.0` `pow( 9.0, 0.5 )` es `3.0`
`sin( x )`	seno trigonométrico de *x* (*x* está en radianes)	`sin( 0.0 )` es `0.0`
`sqrt( x )`	raíz cuadrada de *x*	`sqrt( 900.0 )` es `30.0`
`tan( x )`	tangente trigonométrica de *x* (*x* está en radianes)	`tan( 0.0 )` es `0.0`

Figura 6.2 | Métodos de la clase `Math`.

Constantes `PI` *y* `E` *de la clase* `Math`

La clase `Math` también declara dos campos que representan unas constantes matemáticas de uso común: **`Math.PI`** y **`Math.E`**. La constante `Math.PI` (3.14159265358979323846) es la proporción de la circunferencia de un círculo con su diámetro. La constante `Math.E` (2.7182818284590452354) es el valor de la base para los logaritmos naturales (que se calculan con el método `static log` de la clase `Math`). Estos campos se declaran en la clase `Math` con los modificadores `public`, `final` y `static`. Al hacerlos `public`, otros programadores pueden utilizar estos campos en sus propias clases. Cualquier campo declarado con la palabra clave **`final`** es constante; su valor no puede modificarse después de inicializar el campo. Tanto `PI` como `E` se declaran como `final`, ya que sus valores nunca cambian. Al hacer a estos campos `static`, se puede acceder a ellos mediante el nombre de clase `Math` y un separador de punto (`.`), justo igual que los métodos de la clase `Math`. En la sección 3.5 vimos que cuando cada objeto de una clase mantiene su propia copia de un atributo, el campo que representa a ese atributo se conoce también como variable de instancia: cada objeto de la clase tiene una instancia separada de la variable en memoria. Hay campos para los cuales cada objeto de una clase no tiene una instancia separada de ese campo. Éste es el caso

con los campos `static`, que se conocen también como **variables de clase**. Cuando se crean objetos de una clase que contiene campos `static`, todos los objetos de la clase comparten una copia de los campos `static` de esa clase. En conjunto, las variables de clase (es decir, las variables `static`) y las variables de instancia representan a los campos de una clase. En la sección 8.11 aprenderá más acerca de los campos `static`.

¿Por qué el método `main` *se declara como* `static`*?*

¿Por qué `main` debe declararse como `static`? Cuando se ejecuta la Máquina Virtual de Java (JVM) con el comando `java`, la JVM trata de invocar al método `main` de la clase que usted le especifica; cuando no se han creado objetos de esa clase. Al declarar a `main` como `static`, la JVM puede invocar a `main` sin tener que crear una instancia de la clase. El método `main` se declara con el siguiente encabezado:

```
public static void main( String args[ ] )
```

Cuando usted ejecuta su aplicación, especifica el nombre de su clase como un argumento para el comando `java`, como sigue

```
java NombreClase argumento1 argumento2 …
```

La JVM carga la clase especificada por *NombreClase* y utiliza el nombre de esa clase para invocar al método `main`. En el comando anterior, *NombreClase* es un **argumento de línea de comandos** para la JVM, que le indica cuál clase debe ejecutar. Después del *NombreClase*, también puede especificar una lista de objetos `String` (separados por espacios) como argumentos de línea de comandos, que la JVM pasará a su aplicación. Dichos argumentos pueden utilizarse para especificar opciones (por ejemplo, un nombre de archivo) para ejecutar la aplicación. Como aprenderá en el capítulo 7, Arreglos, su aplicación puede acceder a esos argumentos de línea de comandos y utilizarlos para personalizar la aplicación.

Comentarios adicionales acerca del método `main`

En capítulos anteriores, todas las aplicaciones tenían una clase que sólo contenía a `main`, y posiblemente una segunda clase que `main` utilizaba para crear y manipular objetos. En realidad, cualquier clase puede contener un método `main`. De hecho, cada uno de nuestros ejemplos con dos clases podría haberse implementado como una sola clase. Por ejemplo, en la aplicación de las figuras 5.9 y 5.10, el método `main` (líneas 6 a 16 de la figura 5.10) podría haberse tomado así como estaba, y colocarse en la clase `LibroCalificaciones` (figura 5.9). Después, para ejecutar la aplicación, sólo habría que escribir el comando `java LibroCalificaciones` en la ventana de comandos; los resultados de la aplicación serían idénticos a los de la versión con dos clases. Puede colocar un método `main` en cada clase que declare. La JVM invoca sólo al método `main` en la clase que se utiliza para ejecutar la aplicación. Algunos programadores aprovechan esto para crear un pequeño programa de prueba en cada clase que declaran.

6.4 Declaración de métodos con múltiples parámetros

Los capítulos 3 a 5 presentaron clases que contienen métodos simples, que a lo más tenían un parámetro. A menudo, los métodos requieren más de una pieza de información para realizar sus tareas. Ahora le mostraremos cómo escribir métodos con varios parámetros.

La aplicación de las figuras 6.3 y 6.4 utiliza el método `maximo`, declarado por el programador, para determinar y devolver el mayor de tres valores `double` que introduce el usuario. Cuando la aplicación empieza a ejecutarse, el método `main` de la clase `PruebaBuscadorMaximo` (líneas 7 a 11 de la figura 6.4) crea un objeto de la clase `BuscadorMaximo` (línea 9) y llama al método `determinarMaximo` del objeto (línea 10) para producir los resultados del programa. En la clase `BuscadorMaximo` (figura 6.3), las líneas 14 a18 del método `determinarMaximo` piden al usuario que introduzca tres valores `double`, y después los leen. La línea 21 llama al método `maximo` (declarado en las líneas 28 a 41) para determinar el mayor de los tres valores `double` que se pasan como argumentos para el método. Cuando el método `maximo` devuelve el resultado a la línea 21, el programa asigna el valor de retorno de `maximo` a la variable local `resultado`. Después, la línea 24 imprime el valor máximo. Al final de esta sección, hablaremos sobre el uso del operador + en la línea 24.

Considere la declaración del método `maximo` (líneas 28 a 41). La línea 28 indica que el método devuelve un valor `double`, que el nombre del método es `maximo` y que el método requiere tres parámetros `double` (`x`, `y` y `z`) para realizar su tarea. Cuando un método tiene más de un parámetro, éstos se especifican como una lista separada

```java
// Fig. 6.3: BuscadorMaximo.java
// Método maximo, declarado por el programador.
import java.util.Scanner;

public class BuscadorMaximo
{
   // obtiene tres valores de punto flotante y determina el valor máximo
   public void determinarMaximo()
   {
      // crea objeto Scanner para introducir datos desde la ventana de comandos
      Scanner entrada = new Scanner( System.in );

      // pide y recibe como entrada tres valores de punto flotante
      System.out.print(
         "Escriba tres valores de punto flotante, separados por espacios: " );
      double numero1 = entrada.nextDouble(); // lee el primer valor double
      double numero2 = entrada.nextDouble(); // lee el segundo valor double
      double numero3 = entrada.nextDouble(); // lee el tercer valor double

      // determina el valor máximo
      double resultado = maximo( numero1, numero2, numero3 );

      // muestra el valor máximo
      System.out.println( "El maximo es: " + resultado );
   } // fin del método determinarMaximo

   // devuelve el máximo de sus tres parámetros double
   public double maximo( double x, double y, double z )
   {
      double valorMaximo = x; // asume que x es el mayor para empezar

      // determina si y es mayor que valorMaximo
      if ( y > valorMaximo )
         valorMaximo = y;

      // determina si z es mayor que valorMaximo
      if ( z > valorMaximo )
         valorMaximo = z;

      return valorMaximo;
   } // fin del método maximo
} // fin de la clase BuscadorMaximo
```

Figura 6.3 | Método `maximo`, declarado por el programador, que tiene tres parámetros `double`.

```java
// Fig. 6.4: PruebaBuscadorMaximo.java
// Aplicación para evaluar la clase BuscadorMaximo.

public class PruebaBuscadorMaximo
{
   // punto de inicio de la aplicación
   public static void main( String args[] )
   {
      BuscadorMaximo buscadorMaximo = new BuscadorMaximo();
      buscadorMaximo.determinarMaximo();
   } // fin de main
} // fin de la clase PruebaBuscadorMaximo
```

Figura6.4 | Aplicación para evaluar la clase `BuscadorMaximo`. (Parte 1 de 2).

```
Escriba tres valores de punto flotante, separados por espacios: 9.35 2.74 5.1
El maximo es: 9.35
```

```
Escriba tres valores de punto flotante, separados por espacios: 5.8 12.45 8.32
El maximo es: 12.45
```

```
Escriba tres valores de punto flotante, separados por espacios: 6.46 4.12 10.54
El maximo es: 10.54
```

Figura 6.4 | Aplicación para probar la clase `BuscadorMaximo`. (Parte 2 de 2).

por comas. Cuando se hace la llamada a `maximo` en la línea 21, el parámetro `x` se inicializa con el valor del argumento `numero1`, el parámetro `y` se inicializa con el valor del argumento `numero2` y el parámetro `z` se inicializa con el valor del argumento `numero3`. Debe haber un argumento en la llamada al método para cada parámetro (algunas veces conocido como **parámetro formal**) en la declaración del método. Además, cada argumento debe ser consistente con el tipo del parámetro correspondiente. Por ejemplo, un parámetro de tipo `double` puede recibir valores como 7.35, 22 o –0.03456, pero no objetos `String` como `"hola"`, ni los valores booleanos `true` o `false`. En la sección 6.7 veremos los tipos de argumentos que pueden proporcionarse en la llamada a un método para cada parámetro de un tipo simple.

Para determinar el valor máximo, comenzamos con la suposición de que el parámetro `x` contiene el valor más grande, por lo que la línea 30 declara la variable local `valorMaximo` y la inicializa con el valor del parámetro `x`. Desde luego, es posible que el parámetro `y` o `z` contenga el valor más grande, por lo que debemos comparar cada uno de estos valores con `valorMaximo`. La instrucción `if` en las líneas 33 y 34 determina si `y` es mayor que `valorMaximo`. De ser así, la línea 34 asigna `y` a `valorMaximo`. La instrucción `if` en las líneas 37 y 38 determina si `z` es mayor que `valorMaximo`. De ser así, la línea 38 asigna `z` a `valorMaximo`. En este punto, el mayor de los tres valores reside en `valorMaximo`, por lo que la línea 40 devuelve ese valor a la línea 21. Cuando el control del programa regresa al punto en donde se llamó al método `maximo`, los parámetros `x`, `y` y `z` de `maximo` ya no están accesibles en la memoria. Observe que los métodos pueden devolver a lo máximo un valor, pero el valor devuelto puede ser una referencia a un objeto que contenga muchos valores.

Observe que `resultado` es una variable local en el método `determinarMaximo`, ya que se declara en el bloque que representa el cuerpo del método. Las variables deben declararse como campos de una clase sólo si se requiere su uso en más de un método de la clase, o si el programa debe almacenar sus valores entre las llamadas a los métodos de la clase.



Error común de programación 6.1

Declarar parámetros del mismo tipo para un método, como `float x, y` en vez de `float x, float y` es un error de sintaxis; se requiere un tipo para cada parámetro en la lista de parámetros.



Observación de ingeniería de software 6.5

Un método que tiene muchos parámetros puede estar realizando demasiadas tareas. Considere dividir el método en métodos más pequeños que realicen las tareas separadas. Como lineamiento, trate de ajustar el encabezado del método en una línea, si es posible.

Implementación del método `maximo` mediante la reutilización del método `Math.max`

En la figura 6.2 vimos que la clase `Math` tiene un método `max`, el cual puede determinar el mayor de dos valores. Todo el cuerpo de nuestro método para encontrar el valor máximo podría también implementarse mediante dos llamadas a `Math.max`, como se muestra a continuación:

```
return Math.max( x, Math.max( y, z ) );
```

La primera llamada a `Math.max` especifica los argumentos `x` y `Math.max( y, z )`. Antes de poder llamar a cualquier método, todos sus argumentos deben evaluarse para determinar sus valores. Si un argumento es una llamada

a un método, es necesario realizar esta llamada para determinar su valor de retorno. Por lo tanto, en la instrucción anterior, primero se evalúa `Math.max( y, z )` para determinar el máximo entre `y` y `z`. Después el resultado se pasa como el segundo argumento para la otra llamada a `Math.max`, que devuelve el mayor de sus dos argumentos. Éste es un buen ejemplo de la reutilización de software: buscamos el mayor de los tres valores reutilizando `Math.max`, el cual busca el mayor de dos valores. Observe lo conciso de este código, en comparación con las líneas 30 a 40 de la figura 6.3.

Ensamblado de cadenas mediante la concatenación

Java permite crear objetos `String` mediante el ensamblado de objetos string más pequeños para formar objetos `string` más grandes, mediante el uso del operador + (o del operador de asignación compuesto +=). A esto se le conoce como **concatenación de objetos *string***. Cuando ambos operandos del operador + son objetos `String`, el operador `+` crea un nuevo objeto `String` en el cual los caracteres del operando derecho se colocan al final de los caracteres en el operando izquierdo. Por ejemplo, la expresión `"hola" + "a todos"` crea el objeto `String` `"hola a todos"`.

En la línea 24 de la figura 6.3, la expresión `"El maximo es: " + resultado` utiliza el operador `+` con operandos de tipo `String` y `double`. Cada valor primitivo y cada objeto en Java tienen una representación `String`. Cuando uno de los operandos del operador + es un objeto `String`, el otro se convierte en `String` y después se concatenan los dos. En la línea 24, el valor `double` se convierte en su representación `string` y se coloca al final del objeto `String "El maximo es: "`. Si hay ceros a la derecha en un valor `double`, éstos se descartan cuando el número se convierte en objeto `String`. Por ende, el número 9.3500 se representa como 9.35 en el objeto `String` resultante.

Los valores primitivos que se utilizan en la concatenación de objetos `String` se convierten en objetos `String`. Si un valor `boolean` se concatena con un objeto `String`, el valor `boolean` se convierte en el objeto `String "true"` o `"false"`. Todos los objetos tienen un método llamado `toString` que devuelve una representación `String` del objeto. Cuando se concatena un objeto con un `String`, se hace una llamada implícita al método `toString` de ese objeto para obtener la representación `String` del mismo. En el capítulo 7, Arreglos, aprenderá más acerca del método `toString`.

Cuando se escribe una literal `String` extensa en el código fuente de un programa, algunas veces los programadores prefieren dividir ese objeto `String` en varios objetos `String` más pequeños, para colocarlos en varias líneas de código y mejorar la legibilidad. En este caso, los objetos `String` pueden reensamblarse mediante el uso de la concatenación. En el capítulo 30, Cadenas, caracteres y expresiones regulares, hablaremos sobre los detalles de los objetos `String`.



Error común de programación 6.2

Es un error de sintaxis dividir una literal `String` en varias líneas en un programa. Si una literal `String` no cabe en una línea, divídala en objetos `String` más pequeños y utilice la concatenación para formar la literal `String` deseada.



Error común de programación 6.3

Confundir el operador +, que se utiliza para la concatenación de cadenas, con el operador + que se utiliza para la suma, puede producir resultados extraños. Java evalúa los operandos de un operador de izquierda a derecha. Por ejemplo, si la variable entera `y` tiene el valor 5, la expresión `"y + 2 = " + y + 2` produce la cadena `"y + 2 = 52"`, no `"y + 2 = 7"`, ya que primero el valor de `y` (5) se concatena con la cadena `"y + 2 ="` y después el valor 2 se concatena con la nueva cadena `"y + 2 = 5"` más larga. La expresión `"y + 2 =" + (y + 2)` produce el resultado deseado `"y + 2 = 7"`.

6.5 Notas acerca de cómo declarar y utilizar los métodos

Hay tres formas de llamar a un método:

1. Utilizando el nombre de un método por sí solo para llamar a otro método de la misma clase, como `maximo( numero1, numero2, numero3 )` en la línea 21 de la figura 6.3.

2. Utilizando una variable que contiene una referencia a un objeto, seguida de un punto (`.`) y del nombre del método para llamar a un método del objeto al que se hace referencia, como en la línea 10 de la figura 6.4, `buscadorMaximo.determinarMaximo()`, con lo cual se llama a un método de la clase `BuscadorMaximo` desde el método `main` de `PruebaBuscadorMaximo`.
3. Utilizando el nombre de la clase y un punto (`.`) para llamar a un método `static` de una clase, como `Math.sqrt( 900.0 )` en la sección 6.3.

Observe que un método `static` sólo puede llamar directamente a otros métodos `static` de la misma clase (es decir, usando el nombre del método por sí solo) y solamente puede manipular de manera directa campos `static` en la misma clase. Para acceder a los miembros no `static` de la clase, un método `static` debe usar una referencia a un objeto de esa clase. Recuerde que los métodos `static` se relacionan con una clase como un todo, mientras que los métodos no `static` se asocian con una instancia específica (objeto) de la clase y pueden manipular las variables de instancia de ese objeto. Es posible que existan muchos objetos de una clase al mismo tiempo, cada uno con sus propias copias de las variables de instancia. Suponga que un método `static` invoca a un método no `static` en forma directa. ¿Cómo sabría el método qué variables de instancia manipular de cuál objeto? ¿Qué ocurriría si no existieran objetos de la clase en el momento en el que se invocara el método no `static`? Es evidente que tal situación sería problemática. Por lo tanto, Java no permite que un método `static` acceda directamente a los miembros no `static` de la misma clase.

Existen tres formas de regresar el control a la instrucción que llama a un método. Si el método no devuelve un resultado, el control regresa cuando el flujo del programa llega a la llave derecha de terminación del método, o cuando se ejecuta la instrucción

```
return;
```

si el método devuelve un resultado, la instrucción

```
return expresión;
```

evalúa la *expresión* y después devuelve el resultado al método que hizo la llamada.



Error común de programación 6.4

Declarar un método fuera del cuerpo de la declaración de una clase, o dentro del cuerpo de otro método es un error de sintaxis.



Error común de programación 6.5

Omitir el tipo de valor de retorno en la declaración de un método es un error de sintaxis.



Error común de programación 6.6

Colocar un punto y coma después del paréntesis derecho que encierra la lista de parámetros de la declaración de un método es un error de sintaxis.



Error común de programación 6.7

Volver a declarar el parámetro de un método como una variable local en el cuerpo de ese método es un error de compilación.



Error común de programación 6.8

Olvidar devolver un valor de un método que debe devolver un valor es un error de compilación. Si se especifica un tipo de valor de retorno distinto de `void`, el método debe contener una instrucción `return` que devuelva un valor consistente con el tipo de valor de retorno del método. Devolver un valor de un método cuyo tipo de valor de retorno se haya declarado como `void` es un error de compilación.

6.6 Pila de llamadas a los métodos y registros de activación

Para comprender la forma en que Java realiza las llamadas a los métodos, necesitamos considerar primero una estructura de datos (es decir, una colección de elementos de datos relacionados) conocida como **pila**. Los estudiantes pueden considerar una pila como una analogía de una pila de platos. Cuando se coloca un plato en la pila, por lo general se coloca en la parte superior (lo que se conoce como **meter** el plato en la pila). De manera similar, cuando se extrae un plato de la pila, siempre se extrae de la parte superior (lo que se conoce como **sacar** el plato de la pila). Las pilas se denominan **estructuras de datos "último en entrar, primero en salir"** (**UEPS**; LIFO, por las siglas en inglés de last-in, first-out); el último elemento que se mete (inserta) en la pila es el primero que se saca (extrae) de ella.

Cuando una aplicación llama a un método, el método llamado debe saber cómo regresar al que lo llamó, por lo que la dirección de retorno del método que hizo la llamada se mete en la **pila de ejecución del programa** (también conocida como **pila de llamadas a los métodos**). Si ocurre una serie de llamadas a métodos, las direcciones de retorno sucesivas se meten en la pila, en el orden "último en entrar, primero en salir", para que cada método pueda regresar al que lo llamó.

La pila de ejecución del programa también contiene la memoria para las variables locales que se utilizan en cada invocación de un método, durante la ejecución de un programa. Estos datos, que se almacenan como una porción de la pila de ejecución del programa, se conocen como el **registro de activación** o **marco de pila** de la llamada a un método. Cuando se hace la llamada a un método, el registro de activación para la llamada se mete en la pila de ejecución del programa. Cuando el método regresa al que lo llamó, el registro de activación para esa llamada al método se saca de la pila y esas variables locales ya no son conocidas para el programa. Si una variable local que contiene una referencia a un objeto es la única variable en el programa con una referencia a ese objeto, cuando se saca de la pila el registro de activación que contiene a esa variable local, el programa ya no puede acceder a ese objeto, y la JVM lo eliminará de la memoria en algún momento dado, durante la "recolección de basura". En la sección 8.10 hablaremos sobre la recolección de basura.

Desde luego que la cantidad de memoria en una computadora es finita, por lo que sólo puede utilizarse cierta cantidad de memoria para almacenar los registros de activación en la pila de ejecución del programa. Si ocurren más llamadas a métodos de las que se puedan almacenar sus registros de activación en la pila de ejecución del programa, se produce un error conocido como **desbordamiento de pila**.

6.7 Promoción y conversión de argumentos

Otra característica importante de las llamadas a los métodos es la **promoción de argumentos**: convertir el valor de un argumento al tipo que el método espera recibir en su correspondiente parámetro. Por ejemplo, una aplicación puede llamar al método `sqrt` de `Math` con un argumento entero, aun cuando el método espera recibir un argumento `double` (pero no viceversa, como pronto veremos). La instrucción

```
System.out.println( Math.sqrt( 4 ) );
```

evalúa `Math.sqrt( 4 )` correctamente e imprime el valor `2.0`. La lista de parámetros de la declaración del método hace que Java convierta el valor `int 4` en el valor `double 4.0` antes de pasar ese valor a `sqrt`. Tratar de realizar estas conversiones puede ocasionar errores de compilación, si no se satisfacen las **reglas de promoción** de Java. Las reglas de promoción especifican qué conversiones son permitidas; esto es, qué conversiones pueden realizarse sin perder datos. En el ejemplo anterior de `sqrt`, un `int` se convierte en `double` sin modificar su valor. No obstante, la conversión de un `double` a un `int` trunca la parte fraccionaria del valor `double`; por consecuencia, se pierde parte del valor. La conversión de tipos de enteros largos a tipos de enteros pequeños (por ejemplo, de `long` a `int`) puede también producir valores modificados.

Las reglas de promoción se aplican a las expresiones que contienen valores de dos o más tipos simples, y a los valores de tipos simples que se pasan como argumentos para los métodos. Cada valor se promueve al tipo "más alto" en la expresión. (En realidad, la expresión utiliza una copia temporal de cada valor; los tipos de los valores originales permanecen sin cambios). La figura 6.5 lista los tipos primitivos y los tipos a los cuales se puede promover cada uno de ellos. Observe que las promociones válidas para un tipo dado siempre se realizan a un tipo más alto en la tabla. Por ejemplo, un `int` puede promoverse a los tipos más altos `long`, `float` y `double`.

Al convertir valores a tipos inferiores en la tabla de la figura 6.5, se producirán distintos valores si el tipo inferior no puede representar el valor del tipo superior (por ejemplo, el valor `int` 2000000 no puede representarse como un `short`, y cualquier número de punto flotante con dígitos después de su punto decimal no pueden

representarse en un tipo entero como `long`, `int` o `short`). Por lo tanto, en casos en los que la información puede perderse debido a la conversión, el compilador de Java requiere que utilicemos un operador de conversión (el cual presentamos en la sección 4.9) para forzar explícitamente la conversión; en caso contrario, ocurre un error de compilación. Eso nos permite "tomar el control" del compilador. En esencia decimos, "Sé que esta conversión podría ocasionar pérdida de información, pero para mis fines aquí, eso está bien". Suponga que el método `cuadrado` calcula el cuadrado de un entero y por ende requiere un argumento `int`. Para llamar a `cuadrado` con un argumento `double` llamado `valorDouble`, tendríamos que escribir la llamada al método de la siguiente forma:

```
cuadrado( (int) valorDouble )
```

La llamada a este método convierte explícitamente el valor de `valorDouble` a un entero, para usarlo en el método `cuadrado`. Por ende, si el valor de `valorDouble` es `4.5`, el método recibe el valor `4` y devuelve `16`, no `20.25`.

Tipo	Promociones válidas
`double`	Ninguna
`float`	`double`
`long`	`float` o `double`
`int`	`long`, `float` o `double`
`char`	`int`, `long`, `float` o `double`
`short`	`int`, `long`, `float` o `double` (pero no `char`)
`byte`	`short`, `int`, `long`, `float` o `double` (pero no `char`)
`boolean`	Ninguna (los valores `boolean` no se consideran números en Java)

Figura 6.5 | Promociones permitidas para los tipos primitivos.



Error común de programación 6.9

Convertir un valor de tipo primitivo a otro tipo primitivo puede modificar ese valor, si el nuevo tipo no es una promoción válida. Por ejemplo, convertir un valor de punto flotante a un valor entero puede introducir errores de truncamiento (pérdida de la parte fraccionaria) en el resultado.

6.8 Paquetes de la API de Java

Como hemos visto, Java contiene muchas clases predefinidas que se agrupan en categorías de clases relacionadas, llamadas paquetes. En conjunto, nos referimos a estos paquetes como la Interfaz de programación de aplicaciones de Java (API de Java), o biblioteca de clases de Java.

A lo largo del texto, las declaraciones `import` especifican las clases requeridas para compilar un programa en Java. Por ejemplo, un programa incluye la declaración

```
import java.util.Scanner;
```

para especificar que el programa utiliza la clase `Scanner` del paquete `java.util`. Esto permite a los programadores utilizar el nombre de la clase `Scanner`, en vez de tener que usar el nombre completo calificado de la clase, `java.util.Scanner`, en el código. Uno de los puntos más fuertes de Java es el extenso número de clases en los paquetes de la API de Java. Algunos paquetes clave se describen en la figura 6.6, que representa sólo una pequeña parte de los componentes reutilizables en la API de Java. Mientras esté aprendiendo este lenguaje, invierta una parte de su tiempo explorando las descripciones de los paquetes y las clases en la documentación para la API de Java (`java.sun.com/javase/6/docs/api/`).

El conjunto de paquetes disponibles en Java SE 6 es bastante extenso. Además de los paquetes sintetizados en la figura 6.6, Java SE 6 incluye paquetes para gráficos complejos, interfaces gráficas de usuario avanzadas, impre-

Paquete	Descripción
`java.applet`	El **Paquete Applet de Java** contiene una clase y varias interfaces requeridas para crear applets de Java; programas que se ejecutan en los navegadores Web. (En el capítulo 20, Introducción a las applets de Java, hablaremos sobre las applets; en el capítulo 10, Programación orientada a objetos: polimorfismo, hablaremos sobre las interfaces).
`java.awt`	El **Paquete Abstract Window Toolkit de Java** contiene las clases e interfaces requeridas para crear y manipular GUIs en Java 1.0 y 1.1. En las versiones actuales de Java, se utilizan con frecuencia los componentes de la GUI de Swing, incluidos en los paquetes `javax.swing`. (Algunos elementos del paquete `java.awt` se describen en el capítulo 11, Componentes de la GUI: parte 1, en el capítulo 12, Gráficos y Java 2D™, y en el capítulo 22, Componentes de la GUI: parte 2).
`java.awt.event`	El **Paquete Abstract Window Toolkit Event de Java** contiene clases e interfaces que habilitan el manejo de eventos para componentes de la GUI en los paquetes `java.awt` y `javax.swing`. (Aprenderá más acerca de este paquete en el capítulo 11, Componentes de la GUI: parte 1, y en el capítulo 22, Componentes de la GUI: parte 2).
`java.io`	El **Paquete de Entrada/Salida de Java** contiene clases e interfaces que permiten a los programas recibir datos de entrada y mostrar datos de salida. (Aprenderá más acerca de este paquete en el capítulo 14, Archivos y flujos).
`java.lang`	El **Paquete del Lenguaje Java** contiene clases e interfaces (descritas a lo largo de este texto) requeridas por muchos programas de Java. Este paquete es importado por el compilador en todos los programas, por lo que usted no necesita hacerlo.
`java.net`	El **Paquete de Red de Java** contiene clases e interfaces que permiten a los programas comunicarse mediante redes de computadoras, como Internet. (Aprenderá más acerca de esto en el capítulo 24, Redes).
`java.text`	El **Paquete de Texto de Java** contiene clases e interfaces que permiten a los programas manipular números, fechas, caracteres y cadenas. El paquete proporciona herramientas de internacionalización que permiten la personalización de un programa con respecto a una configuración regional específica (por ejemplo, un programa puede mostrar cadenas en distintos lenguajes, con base en el país del usuario).
`java.util`	El **Paquete de Utilerías de Java** contiene clases e interfaces utilitarias, que permiten acciones como manipulaciones de fecha y hora, procesamiento de números aleatorios (clase `Random`), almacenar y procesar grandes cantidades de datos y descomponer cadenas en piezas más pequeñas llamadas tokens (clase `StringTokenizer`). (Aprenderá más acerca de las características de este paquete en el capítulo 19, Colecciones).
`javax.swing`	El **Paquete de Componentes GUI Swing de Java** contiene clases e interfaces para los componentes de la GUI Swing de Java, los cuales ofrecen soporte para GUIs portables. (Aprenderá más acerca de este paquete en el capítulo 11, Componentes de la GUI: parte 1, y en el capítulo 22, Componentes de la GUI: parte 2).
`javax.swing.event`	El **Paquete Swing Event de Java** contiene clases e interfaces que permiten el manejo de eventos (por ejemplo, responder a los clics del ratón) para los componentes de la GUI en el paquete `javax.swing`. (Aprenderá más acerca de este paquete en el capítulo 11, Componentes de la GUI: parte 1, y en el capítulo 22, Componentes de la GUI: parte 2).

Figura 6.6 | Paquetes de la API de Java (un subconjunto).

sión, redes avanzadas, seguridad, procesamiento de bases de datos, multimedia, accesibilidad (para personas con discapacidades) y muchas otras funciones. Para una visión general de los paquetes en Java SE 6, visite:

`java.sun.com/javase/6/docs/api/overview-summary.html`

Además, muchos otros paquetes están disponibles para descargarse en `java.sun.com`.

Puede localizar información adicional acerca de los métodos de una clase predefinida de Java en la documentación para la API de Java, en `java.sun.com/javase/6/docs/api/`. Cuando visite este sitio, haga clic en el vínculo **Index** para ver un listado en orden alfabético de todas las clases y los métodos en la API de Java. Localice el nombre de la clase y haga clic en su vínculo para ver la descripción en línea de la clase. Haga clic en el vínculo **METHOD** para ver una tabla de los métodos de la clase. Cada método `static` se enlistará con la palabra `"static"` antes del tipo de valor de retorno del método. Para una descripción más detallada acerca de cómo navegar por la documentación para la API de Java, consulte el apéndice J, Uso de la documentación para la API de Java.



Buena práctica de programación 6.2

Es fácil buscar información en la documentación en línea de la API de Java; además proporciona los detalles acerca de cada clase. Al estudiar una clase en este libro, es conveniente que tenga el hábito de buscar la clase en la documentación en línea, para obtener información adicional.

6.9 Ejemplo práctico: generación de números aleatorios

Ahora analizaremos de manera breve una parte divertida de un tipo popular de aplicaciones de la programación: simulación y juegos. En ésta y en la siguiente sección desarrollaremos un programa de juego bien estructurado con varios métodos. El programa utiliza la mayoría de las instrucciones de control presentadas hasta este punto en el libro, e introduce varios conceptos de programación nuevos.

Hay algo en el ambiente de un casino de apuestas que anima a las personas: desde las elegantes mesas de caoba y fieltro para tirar dados, hasta las máquinas tragamonedas. Es el **elemento de azar**, la posibilidad de que la suerte convierta un bolsillo lleno de dinero en una montaña de riquezas. El elemento de azar puede introducirse en un programa mediante un objeto de la clase `Random` (paquete `java.util`), o mediante el método `static` llamado `random`, de la clase `Math`. Los objetos de la clase `Random` pueden producir valores aleatorios de tipo `boolean`, `byte`, `float`, `double`, `int`, `long` y `gaussianos`, mientras que el método `random` de la clase `Math` puede producir sólo valores de tipo `double` en el rango $0.0 \le x < 1.0$, donde x es el valor regresado por el método `random`. En los siguientes ejemplos, usamos objetos de tipo `Random` para producir valores aleatorios.

Se puede crear un nuevo objeto generador de números aleatorios de la siguiente manera:

```
Random numerosAleatorios = new Random();
```

Después, el objeto generador de números aleatorios puede usarse para generar valores `boolean`, `byte`, `float`, `double`, `int`, `long` y `gaussianos`; aquí sólo hablaremos sobre los valores `int` aleatorios. Para obtener más información sobre la clase `Random`, vaya a `java.sun.com/javase/6/docs/api/java/util/Random.html`.

Considere la siguiente instrucción:

```
int valorAleatorio = numerosAleatorios.nextInt();
```

El método `nextInt` de la clase `Random` genera un valor `int` aleatorio en el rango de –2,147,483,648 a +2,147,483,647. Si el método `nextInt` verdaderamente produce valores aleatorios, entonces cualquier valor en ese rango debería tener una oportunidad (o probabilidad) igual de ser elegido cada vez que se llame al método `nextInt`. Los valores devueltos por `nextInt` son en realidad **números seudoaleatorios** (una secuencia de valores producidos por un cálculo matemático complejo). Ese cálculo utiliza la hora actual del día (que, desde luego, cambia constantemente) para **sembrar** el generador de números aleatorios, de tal forma que cada ejecución de un programa produzca una secuencia distinta de valores aleatorios.

El rango de valores producidos directamente por el método `nextInt` es a menudo distinto del rango de valores requeridos en una aplicación de Java particular. Por ejemplo, un programa que simula el lanzamiento de una moneda sólo requiere 0 para "águila" y 1 para "sol". Un programa para simular el tiro de un dado de seis lados requeriría enteros aleatorios en el rango de 1 a 6. Un programa que adivine en forma aleatoria el siguiente tipo de nave espacial (de cuatro posibilidades distintas) que volará a lo largo del horizonte en un videojuego requeriría números aleatorios en el rango de 1 a 4. Para casos como éstos, la clase `Random` cuenta con otra versión del método `nextInt`, que recibe un argumento `int` y devuelve un valor desde 0 hasta (pero sin incluir) el valor del argumento. Por ejemplo, para simular el lanzamiento de monedas, podría utilizar la instrucción

```
int valorAleatorio = numerosAleatorios.nextInt( 2 );
```

que devuelve 0 o 1.

Tirar un dado de seis lados

Para demostrar los números aleatorios, desarrollaremos un programa que simula 20 tiros de un dado de seis lados, y que muestra el valor de cada tiro. Para empezar, usaremos `nextInt` para producir valores aleatorios en el rango de 0 a 5, como se muestra a continuación:

```
cara = numerosAleatorios.nextInt( 6 );
```

El argumento 6 (que se conoce como el **factor de escala**) representa el número de valores únicos que `nextInt` debe producir (en este caso, seis: 0, 1, 2, 3, 4 y 5). A esta manipulación se le conoce como **escalar** el rango de valores producidos por el método `nextInt` de `Random`.

Un dado de seis lados tiene los números del 1 al 6 en sus caras, no del 0 al 5. Por lo tanto, **desplazamos** el rango de números producidos sumando un **valor de desplazamiento** (en este caso, 1) a nuestro resultado anterior, como en

```
cara = 1 + numerosAleatorios.nextInt( 6 );
```

El valor de desplazamiento (`1`) especifica el primer valor en el conjunto deseado de enteros aleatorios. La instrucción anterior asigna a `cara` un entero aleatorio en el rango de 1 a 6.

La figura 6.7 muestra dos resultados de ejemplo, los cuales confirman que los resultados del cálculo anterior son enteros en el rango de 1 a 6, y que cada ejecución del programa puede producir una secuencia distinta de números aleatorios. La línea 3 importa la clase `Random` del paquete `java.util`. La línea 9 crea el objeto `numerosAleatorios` de la clase `Random` para producir valores aleatorios. La línea 16 se ejecuta 20 veces en un ciclo para tirar el dado. La instrucción `if` (líneas 21 y 22) en el ciclo empieza una nueva línea de salida después de cada cinco números.

```
1  // Fig. 6.7: EnterosAleatorios.java
2  // Enteros aleatorios desplazados y escalados.
3  import java.util.Random; // el programa usa la clase Random
4
5  public class EnterosAleatorios
6  {
7     public static void main( String args[] )
8     {
9        Random numerosAleatorios = new Random(); // generador de números aleatorios
10       int cara; // almacena cada entero aleatorio generado
11
12       // itera 20 veces
13       for ( int contador = 1; contador <= 20; contador++ )
14       {
15          // elige entero aleatorio del 1 al 6
16          cara = 1 + numerosAleatorios.nextInt( 6 );
17
18          System.out.printf( "%d  ", cara ); // muestra el valor generado
19
20          // si contador es divisible entre 5, empieza una nueva línea de salida
21          if ( contador % 5 == 0 )
22                System.out.println();
23       } // fin de for
24    } // fin de main
25 } // fin de la clase EnterosAleatorios
```

```
1  5  3  6  2
5  2  6  5  2
4  4  4  2  6
3  1  6  2  2
```

Figura 6.7 | Enteros aleatorios desplazados y escalados. (Parte 1 de 2).

```
6  5  4  2  6
1  2  5  1  3
6  3  2  2  1
6  4  2  6  4
```

Figura 6.7 | Enteros aleatorios desplazados y escalados. (Parte 2 de 2).

Tirar un dado de seis lados 6000 veces

Para mostrar que los números que produce `nextInt` ocurren con una probabilidad aproximadamente igual, simularemos 6000 tiros de un dado con la aplicación de la figura 6.8. Cada entero de 1 a 6 debe aparecer aproximadamente 1000 veces.

Como se muestra en los dos bloques de resultados, al escalar y desplazar los valores producidos por el método `nextInt`, el programa puede simular de manera realista el tiro de un dado de seis lados. La aplicación utiliza instrucciones de control anidadas (la instrucción `switch` está anidada dentro del `for`) para determinar el número de ocurrencias de cada lado del dado. La instrucción `for` (líneas 21 a 47) itera 6000 veces. Durante cada iteración, la línea 23 produce un valor aleatorio del 1 al 6. Después, ese valor se utiliza como la expresión de control (línea 26) de la instrucción `switch` (líneas 26 a 46). Con base en el valor de `cara`, la instrucción `switch` incrementa una de las seis variables contadores durante cada iteración del ciclo. Cuando veamos los arreglos en el capítulo 7, ¡le mostraremos una forma elegante de reemplazar toda la instrucción `switch` en este programa con una sola instrucción! Observe que la instrucción `switch` no tiene un caso `default`, ya que hemos creado una etiqueta `case` para todos los posibles valores que puede producir la expresión en la línea 23. Ejecute el programa varias veces, y observe los resultados. Como verá, cada vez que ejecute el programa, producirá distintos resultados.

```
1  // Fig. 6.8: TirarDado.java
2  // Tirar un dado de seis lados 6000 veces.
3  import java.util.Random;
4
5  public class TirarDado
6  {
7     public static void main( String args[] )
8     {
9        Random numerosAleatorios = new Random(); // generador de números aleatorios
10
11       int frecuencia1 = 0; // cuenta de veces que se tiró 1
12       int frecuencia2 = 0; // cuenta de veces que se tiró 2
13       int frecuencia3 = 0; // cuenta de veces que se tiró 3
14       int frecuencia4 = 0; // cuenta de veces que se tiró 4
15       int frecuencia5 = 0; // cuenta de veces que se tiró 5
16       int frecuencia6 = 0; // cuenta de veces que se tiró 6
17
18       int cara; // almacena el valor que se tiró más recientemente
19
20       // sintetiza los resultados de tirar un dado 6000 veces
21       for ( int tiro = 1; tiro <= 6000; tiro++ )
22       {
23          cara = 1 + numerosAleatorios.nextInt( 6 ); // número del 1 al 6
24
25          // determina el valor del tiro de 1 a 6 e incrementa el contador apropiado
26          switch ( cara )
27          {
28                case 1:
29                      ++frecuencia1; // incrementa el contador de 1s
30                      break;
```

Figura 6.8 | Tirar un dado de seis lados 6000 veces. (Parte 1 de 2).

```
                case 2:
                        ++frecuencia2; // incrementa el contador de 2s
                        break;
                case 3:
                        ++frecuencia3; // incrementa el contador de 3s
                        break;
                case 4:
                        ++frecuencia4; // incrementa el contador de 4s
                        break;
                case 5:
                        ++frecuencia5; // incrementa el contador de 5s
                        break;
                case 6:
                        ++frecuencia6; // incrementa el contador de 6s
                        break; // opcional al final del switch
         } // fin de switch
      } // fin de for

      System.out.println( "Cara\tFrequencia" ); // encabezados de salida
      System.out.printf( "1\t%d\n2\t%d\n3\t%d\n4\t%d\n5\t%d\n6\t%d\n",
         frecuencia1, frecuencia2, frecuencia3, frecuencia4,
         frecuencia5, frecuencia6 );
   } // fin de main
} // fin de la clase TirarDado
```

```
Cara    Frecuencia
1       982
2       1001
3       1015
4       1005
5       1009
6       988
```

```
Cara    Frecuencia
1       1029
2       994
3       1017
4       1007
5       972
6       981
```

Figura 6.8 | Tirar un dado de seis lados 6000 veces. (Parte 2 de 2).

6.9.1 Escalamiento y desplazamiento generalizados de números aleatorios

Anteriormente demostramos la instrucción

```
cara = 1 + numerosAleatorios.nextInt( 6 );
```

la cual simula el tiro de un dado de seis caras. Esta instrucción siempre asigna a la variable `cara` un entero en el rango `1 ≤ cara ≤ 6`. La amplitud de este rango (es decir, el número de enteros consecutivos en el rango) es `6`, y el número inicial en el rango es `1`. Si hacemos referencia a la instrucción anterior, podemos ver que la amplitud del rango se determina en base al número `6` que se pasa como argumento para el método `nextInt` de `Random`, y que el número inicial del rango es el número `1` que se suma a `numerosAleatorios.nextInt( 6 )`. Podemos generalizar este resultado de la siguiente manera:

`numero =` *valorDesplazamiento* `+ numerosAleatorios.nextInt(` *factorEscala* `);`

en donde *valorDesplazamiento* especifica el primer número en el rango deseado de enteros consecutivos y *factorEscala* especifica cuántos números hay en el rango.

También es posible elegir enteros al azar, a partir de conjuntos de valores distintos a los rangos de enteros consecutivos. Por ejemplo, para obtener un valor aleatorio de la secuencia 2, 5, 8, 11 y 14, podríamos utilizar la siguiente instrucción:

```
numero = 2 + 3 * numerosAleatorios.nextInt( 5 );
```

En este caso, `numerosAleatorios.nextInt( 5 )` produce valores en el rango de 0 a 4. Cada valor producido se multiplica por 3 para producir un número en la secuencia 0, 3, 6, 9 y 12. Después sumamos 2 a ese valor para desplazar el rango de valores y obtener un valor de la secuencia 2, 5, 8, 11 y 14. Podemos generalizar este resultado así:

```
numero = valorDesplazamiento +
    diferenciaEntreValores * numerosAleatorios.nextInt( factorEscala );
```

en donde *valorDesplazamiento* especifica el primer número en el rango deseado de valores, *diferenciaEntreValores* representa la diferencia entre números consecutivos en la secuencia y *factorEscala* especifica cuántos números hay en el rango.

6.9.2 Repetitividad de números aleatorios para prueba y depuración

Como mencionamos en la sección 6.9, los métodos de la clase `Random` en realidad generan números seudoaleatorios con base en cálculos matemáticos complejos. Si se llama repetidas veces a cualquiera de los métodos de `Random`, se produce una secuencia de números que parecen ser aleatorios. El cálculo que producen los números seudoaleatorios utiliza la hora del día como **valor de semilla** para cambiar el punto inicial de la secuencia. Cada nuevo objeto `Random` se siembra a sí mismo con un valor basado en el reloj del sistema computacional al momento en que se crea el objeto, con lo cual se permite que cada ejecución de un programa produzca una secuencia distinta de números aleatorios.

Al depurar una aplicación, algunas veces es útil repetir la misma secuencia exacta de números seudoaleatorios durante cada ejecución del programa. Esta repetitividad nos permite probar que la aplicación esté funcionando para una secuencia específica de números aleatorios, antes de evaluar el programa con distintas secuencias de números aleatorios. Cuando la repetitividad es importante, podemos crear un objeto `Random` de la siguiente manera:

```
Random numerosAleatorios = new Random( valorSemilla );
```

El argumento `valorSemilla` (de tipo `long`) siembra el cálculo del número aleatorio. Si se utiliza siempre el mismo valor para `valorSemilla`, el objeto `Random` produce la misma secuencia de números aleatorios. Para establecer la semilla de un objeto `Random` en cualquier momento durante la ejecución de un programa, podemos llamar al método `setSeed` del objeto, como en

```
numerosAleatorios.setSeed( valorSemilla );
```



Tip de prevención de errores 6.2

Mientras un programa esté en desarrollo, cree el objeto `Random` *con un valor de semilla específico para producir una secuencia repetible de números aleatorios cada vez que se ejecute el programa. Si se produce un error lógico, corrija el error y evalúe el programa otra vez con el mismo valor de semilla; esto le permitirá reconstruir la misma secuencia de números aleatorios que produjeron el error. Una vez que se hayan eliminado los errores lógicos, cree el objeto* `Random` *sin utilizar un valor de semilla, para que el objeto* `Random` *genere una nueva secuencia de números aleatorios cada vez que se ejecute el programa.*

6.10 Ejemplo práctico: un juego de probabilidad (introducción a las enumeraciones)

Un juego de azar popular es el juego de dados conocido como "craps", el cual se juega en casinos y callejones por todo el mundo. Las reglas del juego son simples:

> *Un jugador tira dos dados. Cada dado tiene seis caras, las cuales contienen uno, dos, tres cuatro, cinco y seis puntos negros, respectivamente. Una vez que los dados dejan de moverse, se calcula la suma de los*

puntos negros en las dos caras superiores. Si la suma es 7 u 11 en el primer tiro, el jugador gana. Si la suma es 2, 3 o 12 en el primer tiro (llamado "craps"), el jugador pierde (es decir, la "casa" gana). Si la suma es 4, 5, 6, 8, 9 o 10 en el primer tiro, esta suma se convierte en el "punto" del jugador. Para ganar, el jugador debe seguir tirando los dados hasta que salga otra vez "su punto" (es decir, que tire ese mismo valor de punto). El jugador pierde si tira un 7 antes de llegar a su punto.

La aplicación en las figuras 6.9 y 6.10 simula el juego de craps, utilizando varios métodos para definir la lógica del juego. En el método `main` de la clase `PruebaCraps` (figura 6.10), la línea 8 crea un objeto de la clase `Craps` (figura 6.9) y la línea 9 llama a su método `jugar` para iniciar el juego. El método `jugar` (figura 6.9, líneas 21 a 65) llama al método `tirarDado` (figura 6.9, líneas 68 a 81) según sea necesario para tirar los dos dados y calcular su suma. Los cuatro resultados de ejemplo en la figura 6.10 muestran que se ganó en el primer tiro, se perdió en el primer tiro, se ganó en un tiro subsiguiente y se perdió en un tiro subsiguiente, en forma respectiva.

```
1  // Fig. 6.9: Craps.java
2  // La clase Craps simula el juego de dados "craps".
3  import java.util.Random;
4
5  public class Craps
6  {
7     // crea un generador de números aleatorios para usarlo en el método tirarDado
8     private Random numerosAleatorios = new Random();
9
10    // enumeración con constantes que representan el estado del juego
11    private enum Estado { CONTINUA, GANO, PERDIO };
12
13    // constantes que representan tiros comunes del dado
14    private final static int DOS_UNOS = 2;
15    private final static int TRES = 3;
16    private final static int SIETE = 7;
17    private final static int ONCE = 11;
18    private final static int DOCE = 12;
19
20    // ejecuta un juego de craps
21    public void jugar()
22    {
23       int miPunto = 0; // punto si no gana o pierde en el primer tiro
24       Estado estadoJuego; // puede contener CONTINUA, GANO o PERDIO
25
26       int sumaDeDados = tirarDados(); // primer tiro de los dados
27
28       // determina el estado del juego y el punto con base en el primer tiro
29       switch ( sumaDeDados )
30       {
31          case SIETE: // gana con 7 en el primer tiro
32          case ONCE: // gana con 11 en el primer tiro
33             estadoJuego = Estado.GANO;
34             break;
35          case DOS_UNOS: // pierde con 2 en el primer tiro
36          case TRES: // pierde con 3 en el primer tiro
37          case DOCE: // pierde con 12 en el primer tiro
38             estadoJuego = Estado.PERDIO;
39             break;
40          default: // no ganó ni perdió, por lo que guarda el punto
41             estadoJuego = Estado.CONTINUA; // no ha terminado el juego
42             miPunto = sumaDeDados; // guarda el punto
43             System.out.printf( "El punto es %d\n", miPunto );
```

Figura 6.9 | La clase `Craps` simula el juego de dados "craps". (Parte 1 de 2).

```
44               break; // opcional al final del switch
45         } // fin de switch
46 
47         // mientras el juego no esté terminado
48         while ( estadoJuego == Estado.CONTINUA ) // no GANO ni PERDIO
49         {
50            sumaDeDados = tirarDados(); // tira los dados de nuevo
51 
52            // determina el estado del juego
53            if ( sumaDeDados == miPunto ) // gana haciendo un punto
54               estadoJuego = Estado.GANO;
55            else
56               if ( sumaDeDados == SIETE ) // pierde al tirar 7 antes del punto
57                  estadoJuego = Estado.PERDIO;
58         } // fin de while
59 
60         // muestra mensaje de que ganó o perdió
61         if ( estadoJuego == Estado.GANO )
62            System.out.println( "El jugador gana" );
63         else
64            System.out.println( “El jugador pierde" );
65      } // fin del método jugar
66 
67      // tira los dados, calcula la suma y muestra los resultados
68      public int tirarDados()
69      {
70         // elige valores aleatorios para los dados
71         int dado1 = 1 + numerosAleatorios.nextInt( 6 ); // primer tiro del dado
72         int dado2 = 1 + numerosAleatorios.nextInt( 6 ); // segundo tiro del dado
73 
74         int suma = dado1 + dado2; // suma de los valores de los dados
75 
76         // muestra los resultados de este tiro
77         System.out.printf( "El jugador tiro %d + %d = %d\n",
78            dado1, dado2, suma );
79 
80         return suma; // devuelve la suma de los dados
81      } // fin del método tirarDados
82   } // fin de la clase Craps
```

Figura 6.9 | La clase `Craps` simula el juego de dados "craps". (Parte 2 de 2).

Hablaremos sobre la declaración de la clase `Craps` en la figura 6.9. En las reglas del juego, el jugador debe tirar dos dados en el primer tiro y debe hacer lo mismo en todos los tiros subsiguientes. Declaramos el método `tirarDados` (líneas 68 a 81) para tirar el dado y calcular e imprimir su suma. El método `tirarDados` se declara una vez, pero se llama desde dos lugares (líneas 26 y 50) en el método `jugar`, el cual contiene la lógica para un juego completo de craps. El método `tirarDados` no tiene argumentos, por lo cual su lista de parámetros está vacía. Cada vez que se llama, `tirarDados` devuelve la suma de los dados, por lo que se indica el tipo de valor de retorno `int` en el encabezado del método (línea 68). Aunque las líneas 71 y 72 se ven iguales (excepto por el nombre de los dados), no necesariamente producen el mismo resultado. Cada una de estas instrucciones produce un valor aleatorio en el rango de 1 a 6. Observe que `numerosAleatorios` (se utiliza en las líneas 71 y 72) no se declara en el método, sino que se declara como una variable de instancia `private` de la clase y se inicializa en la línea 8. Esto nos permite crear un objeto `Random` que se reutiliza en cada llamada a `tirarDados`.

El juego es razonablemente complejo. El jugador puede ganar o perder en el primer tiro, o puede ganar o perder en cualquier tiro subsiguiente. El método `jugar` (líneas 21 a 65) utiliza a la variable local `miPunto` (línea 23) para almacenar el "punto" si el jugador no gana o pierde en el primer tiro, a la variable local `estadoJuego` (línea 24) para llevar el registro del estado del juego en general y a la variable local `sumaDeDados` (línea 26) para

```
// Fig. 6.10: PruebaCraps.java
// Aplicación para probar la clase Craps.

public class PruebaCraps
{
   public static void main( String args[] )
   {
      Craps juego = new Craps();
      juego.jugar(); // juega un juego de craps
   } // fin de main
} // fin de la clase PruebaCraps
```

```
El jugador tiro 5 + 6 = 11
El jugador gana
```

```
El jugador tiro 1 + 2 = 3
El jugador pierde
```

```
El jugador tiro 5 + 4 = 9
El punto es 9
El jugador tiro 2 + 2 = 4
El jugador tiro 2 + 6 = 8
El jugador tiro 4 + 2 = 6
El jugador tiro 3 + 6 = 9
El jugador gana
```

```
El jugador tiro 2 + 6 = 8
El punto es 8
El jugador tiro 5 + 1 = 6
El jugador tiro 2 + 1 = 3
El jugador tiro 1 + 6 = 7
El jugador pierde
```

Figura 6.10 | Aplicación para probar la clase `Craps`.

almacenar la suma de los dados para el tiro más reciente. Observe que `miPunto` se inicializa con `0` para asegurar que la aplicación se compile. Si no inicializa `miPunto`, el compilador genera un error ya que `miPunto` no recibe un valor en todas las etiquetas `case` de la instrucción `switch` y, en consecuencia, el programa podría tratar de utilizar `miPunto` antes de que se le asigne un valor. En contraste, `estadoJuego` no requiere inicialización, ya que *se* le asigna un valor en cada etiqueta `case` de la instrucción `switch`; por lo tanto, se garantiza que se inicialice antes de usarse.

Observe que la variable local `estadoJuego` (línea 24) se declara como de un nuevo tipo llamado `Estado`, el cual declaramos en la línea 11. El tipo `Estado` se declara como un miembro `private` de la clase `Craps`, ya que sólo se utiliza en esa clase. `Estado` es un tipo declarado por el programador, denominado **enumeración**, que en su forma más simple declara un conjunto de constantes representadas por identificadores. Una enumeración es un tipo especial de clase, que se introduce mediante la palabra clave `enum` y un nombre para el tipo (en este caso, `Estado`). Al igual que con una clase, las llaves (`{` y `}`) delimitan el cuerpo de una declaración de `enum`. Dentro de las llaves hay una lista, separada por comas, de **constantes de enumeración**, cada una de las cuales representa un valor único. Los identificadores en una `enum` deben ser únicos (en el capítulo 8 aprenderá más acerca de las enumeraciones).



Buena práctica de programación 6.3

Use sólo letras mayúsculas en los nombres de las constantes de enumeración. Esto hace que resalten y le recuerdan que las constantes de enumeración no son variables.

A las variables de tipo `Estado` se les debe asignar sólo una de las tres constantes declaradas en la enumeración (línea 11), o se producirá un error de compilación. Cuando el jugador gana el juego, el programa asigna a la variable local `estadoJuego` el valor `Estado.GANO` (líneas 33 y 54). Cuando el jugador pierde el juego, la aplicación asigna a la variable local `estadoJuego` el valor `Estado.PERDIO` (líneas 38 y 57). En cualquier otro caso, el programa asigna a la variable local `estadoJuego` el valor `Estado.CONTINUA` (línea 41) para indicar que el juego no ha terminado y hay que tirar los dados otra vez.



Buena práctica de programación 6.4

El uso de constantes de enumeración (como `Estado.GANO`, `Estado.PERDIO` y `Estado.CONTINUA`) en vez de valores enteros literales (como 0, 1 y 2) puede hacer que los programas sean más fáciles de leer y de mantener.

La línea 26 en el método `jugar` llama a `tirarDados`, el cual elige dos valores aleatorios del 1 al 6, muestra el valor del primer dado, el del segundo y la suma de los dos dados, y devuelve esa suma. Después el método `jugar` entra a la instrucción `switch` en las líneas 29 a 45, que utiliza el valor de `sumaDeDados` de la línea 26 para determinar si el jugador ganó o perdió el juego, o si debe continuar con otro tiro. Las sumas de los dados que ocasionan que se gane o pierda el juego en el primer tiro se declaran como constantes `public final static int` en las líneas 14 a 18. Estos valores se utilizan en las etiquetas `case` de la instrucción `switch`. Los nombres de los identificadores utilizan los términos comunes en el casino para estas sumas. Observe que estas constantes, al igual que las constantes `enum`, se declaran todas con letras mayúsculas por convención, para que resalten en el programa. Las líneas 31 a 34 determinan si el jugador ganó en el primer tiro con `SIETE` (7) u `ONCE` (11). Las líneas 35 a 39 determinan si el jugador perdió en el primer tiro con `DOS_UNOS` (2), `TRES` (3) o `DOCE` (12). Después del primer tiro, si el juego no se ha terminado, el caso `default` (líneas 40 a 44) establece `estadoJuego` en `Estado.CONTINUA`, guarda `sumaDeDados` en `miPunto` y muestra el punto.

Si aún estamos tratando de "hacer nuestro punto" (es decir, el juego continúa de un tiro anterior), se ejecuta el ciclo de las líneas 48 a 58. En la línea 50 se tira el dado otra vez. Si `sumaDeDados` concuerda con `miPunto` en la línea 53, la línea 54 establece `estadoJuego` en `Estado.GANO` y el ciclo termina, ya que el juego está terminado. En la línea 56, si `sumaDeDados` es igual a `SIETE` (7), la línea 57 asigna el valor `Estado.PERDIO` a `estadoJuego` y el ciclo termina, ya que se acabó el juego. Cuando termina el juego, las líneas 61 a 64 muestran un mensaje en el que se indica si el jugador ganó o perdió, y el programa termina.

Observe el uso de varios mecanismos de control del programa que hemos visto antes. La clase `Craps`, en conjunto con la clase `PruebaCraps`, utiliza tres métodos: `main`, `jugar` (que se llama desde `main`) y `tirarDados` (que se llama dos veces desde `jugar`), y las instrucciones de control `switch`, `while`, `if…else` e `if` anidado. Observe también el uso de múltiples etiquetas `case` en la instrucción `switch` para ejecutar las mismas instrucciones para las sumas de `SIETE` y `ONCE` (líneas 31 y 32), y para las sumas de `DOS_UNOS`, `TRES` y `DOCE` (líneas 35 a 37).

Tal vez se esté preguntando por qué declaramos las sumas de los dados como constantes `public final static int` en vez de constantes `enum`. La respuesta está en el hecho de que el programa debe comparar la variable `int` llamada `sumaDeDados` (línea 26) con estas constantes para determinar el resultado de cada tiro. Suponga que declararemos constantes que contengan `enum Suma` (por ejemplo, `Suma.DOS_UNOS`) para representar las cinco sumas utilizadas en el juego, y que después usaremos estas constantes en las etiquetas `case` de la instrucción `switch` (líneas 29 a 45). Hacer esto evitaría que pudiéramos usar `sumaDeDados` como la expresión de control de la instrucción `switch`, ya que Java no permite que un `int` se compare con una constante de enumeración. Para lograr la misma funcionalidad que el programa actual, tendríamos que utilizar una variable `sumaActual` de tipo `Suma` como expresión de control para el `switch`. Por desgracia, Java no proporciona una manera fácil de convertir un valor `int` en una constante `enum` específica. Podríamos traducir un `int` en una constante `enum` mediante una instrucción `switch` separada. Sin duda, esto sería complicado y no mejoraría la legibilidad del programa (lo cual echaría a perder el propósito de usar una `enum`).

6.11 Alcance de las declaraciones

Ya hemos visto declaraciones de varias entidades de Java como las clases, los métodos, las variables y los parámetros. Las declaraciones introducen nombres que pueden utilizarse para hacer referencia a dichas entidades de Java. El **alcance** de una declaración es la porción del programa que puede hacer referencia a la entidad declarada por su nombre. Se dice que dicha entidad está "dentro del alcance" para esa porción del programa. En esta sección introduciremos varias cuestiones importantes relacionadas con el alcance. (Para obtener más información sobre

el alcance, consulte la *Especificación del lenguaje Java, sección 6.3: Alcance de una declaración*, en `java.sun.com/docs/books/jls/second_edition/html/names.doc.html#103228`).

Las reglas básicas de alcance son las siguientes:

1. El alcance de la declaración de un parámetro es el cuerpo del método en el que aparece la declaración.
2. El alcance de la declaración de una variable local es a partir del punto en el cual aparece la declaración, hasta el final de ese bloque.
3. El alcance de la declaración de una variable local que aparece en la sección de inicialización del encabezado de una instrucción `for` es el cuerpo de la instrucción `for` y las demás expresiones en el encabezado.
4. El alcance de un método o campo de una clase es todo el cuerpo de la clase. Esto permite a los métodos no `static` de la clase utilizar cualquiera de los campos y otros métodos de la clase.

Cualquier bloque puede contener declaraciones de variables. Si una variable local o parámetro en un método tiene el mismo nombre que un campo, el campo se "oculta" hasta que el bloque termina su ejecución; a esto se le llama **ocultación de variables (shadowing)**. En el capítulo 8 veremos cómo acceder a los campos ocultos.



Error común de programación 6.10

Cuando una variable local se declara más de una vez en un método, se produce un error de compilación.



Tip de prevención de errores 6.3

Use nombres distintos para los campos y las variables locales, para ayudar a evitar los errores lógicos sutiles que se producen cuando se hace la llamada a un método y una variable local de ese método oculta un campo con el mismo nombre en la clase.

La aplicación en las figuras 6.11 y 6.12 demuestra las cuestiones de alcance con los campos y las variables locales. Cuando la aplicación empieza a ejecutarse, el método `main` de la clase `PruebaAlcance` (figura 6.12, líneas 7 a 11) crea un objeto de la clase `Alcance` (línea 9) y llama al método `iniciar` del objeto (línea 10) para producir el resultado de la aplicación (el cual se muestra en la figura 6.12).

```
// Fig. 6.11: Alcance.java
// La clase Alcance demuestra los alcances de los campos y las variables locales.

public class Alcance
{
   // campo accesible para todos los métodos de esta clase
   private int x = 1;

   // el método iniciar crea e inicializa la variable local x
   // y llama a los métodos usarVariableLocal y usarCampo
   public void iniciar()
   {
      int x = 5; // la variable local x del método oculta al campo x

      System.out.printf( "la x local en el metodo iniciar es %d\n", x );

      usarVariableLocal(); // usarVariableLocal tiene la x local
      usarCampo(); // usarCampo usa el campo x de la clase Alcance
      usarVariableLocal(); // usarVariableLocal reinicia a la x local
      usarCampo(); // el campo x de la clase Alcance retiene su valor

```

Figura 6.11 | La clase `Alcance` demuestra los alcances de los campos y las variables locales. (Parte 1 de 2).

```
      System.out.printf( "\nla x local en el metodo iniciar es %d\n”, x );
   } // fin del método iniciar

   // crea e inicializa la variable local x durante cada llamada
   public void usarVariableLocal()
   {
      int x = 25; // se inicializa cada vez que se llama a usarVariableLocal

      System.out.printf(
         "\nla x local al entrar al metodo usarVariableLocal es %d\n", x );
      ++x; // modifica la variable x local de este método
      System.out.printf(
         "la x local antes de salir del metodo usarVariableLocal es %d\n", x );
   } // fin del método usarVariableLocal

   // modifica el campo x de la clase Alcance durante cada llamada
   public void usarCampo()
   {
      System.out.printf(
         "\nel campo x al entrar al metodo usarCampo es %d\n", x );
      x *= 10; // modifica el campo x de la clase Alcance
      System.out.printf(
         "el campo x antes de salir del metodo usarCampo es %d\n", x );
   } // fin del método usarCampo
} // fin de la clase Alcance
```

Figura 6.11 | La clase `Alcance` demuestra los alcances de los campos y las variables locales. (Parte 2 de 2).

```
// Fig. 6.12: PruebaAlcance.java
// Aplicación para probar la clase Alcance.

public class PruebaAlcance
{
   // punto inicial de la aplicación
   public static void main( String args[] )
   {
      Alcance alcancePrueba = new Alcance();
      alcancePrueba.iniciar();
   } // fin de main
} // fin de la clase PruebaAlcance
```

```
la x local en el metodo iniciar es 5

la x local al entrar al metodo usarVariableLocal es 25
la x local antes de salir del metodo usarVariableLocal es 26

el campo x al entrar al metodo usarCampo es 1
el campo x antes de salir del metodo usarCampo es 10

la x local al entrar al metodo usarVariableLocal es 25
la x local antes de salir del metodo usarVariableLocal es 26

el campo x al entrar al metodo usarCampo es 10
el campo x antes de salir del metodo usarCampo es 100

la x local en el metodo iniciar es 5
```

Figura 6.12 | Aplicación para probar la clase `Alcance`.

En la clase `Alcance`, la línea `7` declara e inicializa el campo `x` en `1`. Este campo se oculta en cualquier bloque (o método) que declare una variable local llamada `x`. El método `iniciar` (líneas 11 a 23) declara una variable local `x` (línea 13) y la inicializa en 5. El valor de esta variable local se imprime para mostrar que el campo `x` (cuyo valor es `1`) se oculta en el método `iniciar`. El programa declara otros dos métodos: `usarVariableLocal` (líneas 26 a 35) y `usarCampo` (líneas 38 a 45); cada uno de ellos no tiene argumentos y no devuelve resultados. El método `iniciar` llama a cada método dos veces (líneas 17 a 20). El método `usarVariableLocal` declara la variable local `x` (línea 28). Cuando se llama por primera vez a `usarVariableLocal` (línea 17), crea una variable local `x` y la inicializa en 25 (línea 28), muestra en pantalla el valor de `x` (líneas 30 y 31), incrementa `x` (línea 32) y muestra en pantalla el valor de `x` otra vez (líneas 33 y 34). Cuando se llama a `usarVariableLocal` por segunda vez (línea 19), vuelve a crear la variable local `x` y la reinicializa con `25`, por lo que la salida de cada llamada a `usarVariableLocal` es idéntica.

El método `usarCampo` no declara variables locales. Por lo tanto, cuando hace referencia a `x`, se utiliza el campo `x` (línea 7) de la clase. Cuando el método `usarCampo` se llama por primera vez (línea 18), muestra en pantalla el valor (`1`) del campo `x` (líneas 40 y 41), multiplica el campo `x` por 10 (línea 42) y muestra en pantalla el valor (`10`) del campo `x` otra vez (líneas 43 y 44) antes de regresar. La siguiente vez que se llama al método `usarCampo` (línea 20), el campo `x` tiene el valor modificado de `10`, por lo que el método muestra en pantalla un `10` y después un `100`. Por último, en el método `iniciar` el programa muestra en pantalla el valor de la variable local `x` otra vez (línea 22), para mostrar que ninguna de las llamadas a los métodos modificó la variable local `x` de `iniciar`, ya que todos los métodos hicieron referencia a las variables llamadas `x` en otros alcances.

6.12 Sobrecarga de métodos

Pueden declararse métodos con el mismo nombre en la misma clase, siempre y cuando tengan distintos conjuntos de parámetros (determinados en base al número, tipos y orden de los parámetros). A esto se le conoce como **sobrecarga de métodos**. Cuando se hace una llamada a un método sobrecargado, el compilador de Java selecciona el método apropiado mediante un análisis del número, tipos y orden de los argumentos en la llamada. Por lo general, la sobrecarga de métodos se utiliza para crear varios métodos con el mismo nombre que realicen la misma tarea o tareas similares, pero con distintos tipos o distintos números de argumentos. Por ejemplo, los métodos `abs`, `min` y `max` de `Math` (sintetizados en la sección 6.3) se sobrecargan con cuatro versiones cada uno:

1. Uno con dos parámetros `double`.
2. Uno con dos parámetros `float`.
3. Uno con dos parámetros `int`.
4. Uno con dos parámetros `long`.

Nuestro siguiente ejemplo demuestra cómo declarar e invocar métodos sobrecargados. En el capítulo 8 presentaremos ejemplos de constructores sobrecargados.

Declaración de métodos sobrecargados

En nuestra clase `SobrecargaMetodos` (figura 6.13) incluimos dos versiones sobrecargadas de un método llamado `cuadrado`: una que calcula el cuadrado de un `int` (y devuelve un `int`) y otra que calcula el cuadrado de un `double` (y devuelve un `double`). Aunque estos métodos tienen el mismo nombre, además de listas de parámetros y cuerpos similares, podemos considerarlos simplemente como métodos *diferentes*. Puede ser útil si consideramos los nombres de los métodos como "`cuadrado de int`" y "`cuadrado de double`", respectivamente. Cuando la aplicación empieza a ejecutarse, el método `main` de la clase `PruebaSobrecargaMetodos` (figura 6.14, líneas 6 a 10) crea un objeto de la clase `SobrecargaMetodos` (línea 8) y llama al método `probarMetodosSobrecargados` del objeto (línea 9) para producir la salida del programa (figura 6.14).

En la figura 6.13, la línea 9 invoca al método `cuadrado` con el argumento `7`. Los valores enteros literales se tratan como de tipo `int`, por lo que la llamada al método en la línea 9 invoca a la versión de `cuadrado` de las líneas 14 a 19, la cual especifica un parámetro `int`. De manera similar, la línea 10 invoca al método `cuadrado` con el argumento `7.5`. Los valores de las literales de punto flotante se tratan como de tipo `double`, por lo que la llamada al método en la línea 10 invoca a la versión de `cuadrado` de las líneas 22 a 27, la cual especifica un parámetro `double`. Cada método imprime en pantalla primero una línea de texto, para mostrar que se llamó al método

apropiado en cada caso. Observe que los valores en las líneas 10 y 24 se muestran con el especificador de formato `%f` y que no especificamos una precisión en ninguno de los dos casos. De manera predeterminada, los valores de punto flotante se muestran con seis dígitos de precisión, si ésta no se especifica en el especificador de formato.

```
1  // Fig. 6.13: SobrecargaMetodos.java
2  // Declaraciones de métodos sobrecargados.
3
4  public class SobrecargaMetodos
5  {
6     // prueba los métodos cuadrado sobrecargados
7     public void probarMetodosSobrecargados()
8     {
9        System.out.printf( "El cuadrado del entero 7 es %d\n", cuadrado( 7 ) );
10       System.out.printf( "El cuadrado del double 7.5 es %f\n", cuadrado( 7.5 ) );
11    } // fin del método probarMetodosSobrecargados
12
13    // método cuadrado con argumento int
14    public int cuadrado( int valorInt )
15    {
16       System.out.printf( "\nSe llamo a cuadrado con argumento int: %d\n",
17          valorInt );
18       return valorInt * valorInt;
19    } // fin del método cuadrado con argumento int
20
21    // método cuadrado con argumento double
22    public double cuadrado( double valorDouble )
23    {
24       System.out.printf( "\nSe llamo a cuadrado con argumento double: %f\n",
25          valorDouble );
26       return valorDouble * valorDouble;
27    } // fin del método cuadrado con argumento double
28 } // fin de la clase SobrecargaMetodos
```

Figura 6.13 | Declaraciones de métodos sobrecargados.

```
// Fig. 6.14: PruebaSobrecargaMetodos.java
// Aplicación para probar la clase SobrecargaMetodos.

public class PruebaSobrecargaMetodos
{
   public static void main( String args[] )
   {
      SobrecargaMetodos sobrecargaMetodos = new SobrecargaMetodos();
      sobrecargaMetodos.probarMetodosSobrecargados();
   } // fin de main
} // fin de la clase PruebaSobrecargaMetodos
```

```
Se llamo a cuadrado con argumento int: 7
El cuadrado del entero 7 es 49

Se llamo a cuadrado con argumento double: 7.500000
El cuadrado del double 7.5 es 56.250000
```

Figura 6.14 | Aplicación para probar la clase `SobrecargaMetodos`.

Cómo se diferencian los métodos sobrecargados entre sí

El compilador diferencia los métodos sobrecargados en base a su **firma**: una combinación del nombre del método y del número, tipos y orden de sus parámetros. Si el compilador sólo se fijara en los nombres de los métodos durante la compilación, el código de la figura 6.13 sería ambiguo; el compilador no sabría cómo distinguir entre los dos métodos `cuadrado` (líneas 14 a 19 y 22 a 27). De manera interna, el compilador utiliza nombres de métodos más largos que incluyen el nombre del método original, el tipo de cada parámetro y el orden exacto de los parámetros para determinar si los métodos en una clase son únicos en esa clase.

Por ejemplo, en la figura 6.13 el compilador podría utilizar el nombre lógico "`cuadrado` de `int`" para el método `cuadrado` que especifica un parámetro `int`, y el método "`cuadrado` de `double`" para el método `cuadrado` que especifica un parámetro `double` (los nombres reales que utiliza el compilador son más complicados). Si la declaración de `metodo1` empieza así:

```
void metodo1( int a, float b )
```

entonces el compilador podría usar el nombre lógico "`metodo1` de `int` y `float`". Si los parámetros se especificaran así:

```
void metodo1( float a, int b )
```

entonces el compilador podría usar el nombre lógico "`metodo1` de `float` e `int`". Observe que el orden de los tipos de los parámetros es importante; el compilador considera que los dos encabezados anteriores de `metodo1` son distintos.

Tipos de valores de retorno de los métodos sobrecargados

Al hablar sobre los nombres lógicos de los métodos que utiliza el compilador, no mencionamos los tipos de valores de retorno de los métodos. Esto se debe a que las *llamadas* a los métodos no pueden diferenciarse en base al tipo de valor de retorno. El programa de la figura 6.15 ilustra los errores que genera el compilador cuando dos métodos tienen la misma firma, pero distintos tipos de valores de retorno. Los métodos sobrecargados pueden tener tipos de valor de retorno distintos si los métodos tienen distintas listas de parámetros. Además, los métodos sobrecargados no necesitan tener el mismo número de parámetros.

```
// Fig. 6.15: SobrecargaMetodos.java
// Los métodos sobrecargados con firmas idénticas producen errores de
// compilación, aun si los tipos de valores de retorno son distintos.

public class ErrorSobrecargaMetodos
{
   // declaración del método cuadrado con argumento int
   public int cuadrado( int x )
   {
      return x * x;
   }

   // la segunda declaración del método cuadrado con argumento int produce un error
   // de compilación, aun cuando los tipos de valores de retorno son distintos
   public double cuadrado( int y )
   {
      return y * y;
   }
} // fin de la clase ErrorSobrecargaMetodos
```

```
ErrorSobrecargaMetodos.java:15: cuadrado(int) is already defined in ErrorSobrecargaMetodos
    public double cuadrado( int y )
                  ^
1 error
```

Figura 6.15 | Las declaraciones de métodos sobrecargados con firmas idénticas producen errores de compilación, aun si los tipos de valores de retorno son distintos.



Error común de programación 6.11

Declarar métodos sobrecargados con listas de parámetros idénticas es un error de compilación, sin importar que los tipos de los valores de retorno sean distintos.

6.13 (Opcional) Ejemplo práctico de GUI y gráficos: colores y figuras rellenas

Aunque podemos crear muchos diseños interesantes sólo con líneas y figuras básicas, la clase `Graphics` proporciona muchas herramientas más. Las siguientes dos herramientas que presentaremos son los colores y las figuras rellenas. El color agrega otra dimensión a los dibujos que ve un usuario en la pantalla de la computadora. Las figuras rellenas cubren regiones completas con colores sólidos, en vez de dibujar sólo contornos.

Los colores que se muestran en las pantallas de las computadoras se definen en base a sus componentes rojo, verde y azul. Estos componentes, llamados **valores RGB**, tienen valores enteros de 0 a 255. Entre más alto sea el valor de un componente específico, más intensidad de color tendrá esa figura. Java utiliza la clase **`Color`** en el paquete `java.awt` para representar colores usando sus valores RGB. Por conveniencia, el objeto `Color` contiene 13 objetos `static Color` predefinidos: `Color.BLACK`, `Color.BLUE`, `Color.CYAN`, `Color.DARK_GRAY`, `Color.GRAY`, `Color.GREEN`, `Color.LIGHT_GRAY`, `Color.MAGENTA`, `Color.ORANGE`, `Color.PINK`, `Color.RED`, `Color.WHITE` y `Color.YELLOW`. La clase `Color` también contiene un constructor de la forma:

```
public Color( int r, int g, int b )
```

de manera que podemos crear colores específicos, con sólo especificar valores para los componentes individuales rojo, verde y azul de un color.

Los rectángulos y los óvalos rellenos se dibujan usando los métodos **`fillRect`** y **`fillOval`** de `Graphics`, respectivamente. Estos dos métodos tienen los mismos parámetros que sus contrapartes `drawRect` y `drawOval` sin relleno: los primeros dos parámetros son las coordenadas para la esquina superior izquierda de la figura, mientras que los otros dos parámetros determinan su anchura y su altura. El ejemplo de las figuras 6.16 y 6.17 demuestra los colores y las figuras rellenas, al dibujar y mostrar una cara sonriente amarilla (esto lo verá en su pantalla).

```
// Fig. 6.16: DibujarCaraSonriente.java
// Demuestra las figuras rellenas.
import java.awt.Color;
import java.awt.Graphics;
import javax.swing.JPanel;

public class DibujarCaraSonriente extends JPanel
{
   public void paintComponent( Graphics g )
   {
      super.paintComponent( g );

      // dibuja la cara
      g.setColor( Color.YELLOW );
      g.fillOval( 10, 10, 200, 200 );

      // dibuja los ojos
      g.setColor( Color.BLACK );
      g.fillOval( 55, 65, 30, 30 );
      g.fillOval( 135, 65, 30, 30 );

      // dibuja la boca
      g.fillOval( 50, 110, 120, 60 );

      // convierte la boca en una sonrisa
```

Figura 6.16 | Cómo dibujar una cara sonriente, usando colores y figuras rellenas. (Parte 1 de 2).

```
26          g.setColor( Color.YELLOW );
27          g.fillRect( 50, 110, 120, 30 );
28          g.fillOval( 50, 120, 120, 40 );
29       } // fin del método paintComponent
30    } // fin de la clase DibujarCaraSonriente
```

Figura 6.16 | Cómo dibujar una cara sonriente, usando colores y figuras rellenas. (Parte 2 de 2).

```
// Fig. 6.17: PruebaDibujarCaraSonriente.java
// Aplicación de prueba que muestra una cara sonriente.
import javax.swing.JFrame;

public class PruebaDibujarCaraSonriente
{
   public static void main( String args[] )
   {
      DibujarCaraSonriente panel = new DibujarCaraSonriente();
      JFrame aplicacion = new JFrame();

      aplicacion.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
      aplicacion.add( panel );
      aplicacion.setSize( 230, 250 );
      aplicacion.setVisible( true );
   } // fin de main
} // fin de la clase PruebaDibujarCaraSonriente
```



Figura 6.17 | Creación de un objeto `JFrame` para mostrar una cara sonriente.

Las instrucciones `import` en las líneas 3 a 5 de la figura 6.16 importan las clases `Color`, `Graphics` y `JPanel`. La clase `DibujarCaraSonriente` (líneas 7 a 30) utiliza la clase `Color` para especificar los colores, y utiliza la clase `Graphics` para dibujar. La clase `JPanel` proporciona de nuevo el área en la que vamos a dibujar. La línea 14 en el método `paintComponent` utiliza el método **`setColor`** de `Graphics` para establecer el color actual para dibujar en `Color.YELLOW`. El método `setColor` requiere un argumento, el `Color` a establecer como el color para dibujar. En este caso, utilizamos el objeto predefinido `Color.YELLOW`. La línea 15 dibuja un círculo con un diámetro de 200 para representar la cara; cuando los argumentos anchura y altura son idénticos, el método `fillOval` dibuja un círculo. A continuación, la línea 18 establece el color en `Color.BLACK`, y las líneas 19 y 20 dibujan los ojos. La línea 23 dibuja la boca como un óvalo, pero esto no es exactamente lo que queremos. Para crear una cara feliz, vamos a "retocar" la boca. La línea 26 establece el color en `Color.YELLOW`, de manera que cualquier figura que dibujemos se mezcle con la cara. La línea 27 dibuja un rectángulo con la mitad de altura que la boca. Esto "borra" la mitad superior de la boca, dejando sólo la mitad inferior. Para crear una mejor sonrisa, la línea 28 dibuja otro óvalo para cubrir ligeramente la porción superior de la boca. La clase `PruebaDibujarCaraSonriente` (figura 6.17) crea y muestra un objeto `JFrame` que contiene el dibujo. Cuando se muestra el objeto `JFrame`, el sistema llama al método `paintComponent` para dibujar la cara sonriente.

Ejercicios del ejemplo práctico de GUI y gráficos

6.1 Usando el método `fillOval`, dibuje un tiro al blanco que alterne entre dos colores aleatorios, como en la figura 6.18. Use el constructor `Color( int r, int g, int b )` con argumentos aleatorios para generar colores aleatorios.

6.2 Cree un programa para dibujar 10 figuras rellenas al azar en colores, posiciones y tamaños aleatorios (figura 6.19). El método `paintComponent` debe contener un ciclo que itere 10 veces. En cada iteración, el ciclo debe determinar si se dibujará un rectángulo o un óvalo relleno, crear un color aleatorio y elegir las coordenadas y las medidas al azar. Las coordenadas deben elegirse con base en la anchura y la altura del panel. Las longitudes de los lados deben limitarse a la mitad de la anchura o altura de la ventana.



Figura 6.18 | Un tiro al blanco con dos colores alternantes al azar.



Figura 6.19 | Figuras generadas al azar.

6.14 (Opcional) Ejemplo práctico de Ingeniería de Software: identificación de las operaciones de las clases

En las secciones del Ejemplo práctico de Ingeniería de Software al final de los capítulos 3 a 5, llevamos a cabo los primeros pasos en el diseño orientado a objetos de nuestro sistema ATM. En el capítulo 3 identificamos las clases que necesitaremos implementar, y creamos nuestro primer diagrama de clases. En el capítulo 4 describimos varios atributos de nuestras clases. En el capítulo 5 examinamos los estados de nuestros objetos y modelamos sus transiciones de estado y actividades. En esta sección determinaremos algunas de las operaciones (o comportamientos) de las clases que son necesarias para implementar el sistema ATM.

Identificar las operaciones

Una operación es un servicio que proporcionan los objetos de una clase a los clientes (usuarios) de esa clase. Considere las operaciones de algunos objetos reales. Las operaciones de un radio incluyen el sintonizar su estación y ajustar su volumen (que, por lo general, lo hace una persona que ajusta los controles del radio). Las operaciones de un automóvil incluyen acelerar (operación invocada por el conductor cuando oprime el pedal del acelerador), desacelerar (operación invocada por el conductor cuando oprime el pedal del freno o cuando suelta el pedal del acelerador), dar vuelta y cambiar velocidades. Los objetos de software también pueden ofrecer operaciones; por ejemplo, un objeto de gráficos de software podría ofrecer operaciones para dibujar un círculo, dibujar una línea, dibujar un cuadrado, etcétera. Un objeto de software de hoja de cálculo podría ofrecer operaciones como imprimir la hoja de cálculo, totalizar los elementos en una fila o columna, y graficar la información de la hoja de cálculo como un gráfico de barras o de pastel.

Podemos derivar muchas de las operaciones de cada clase mediante un análisis de los verbos y las frases verbales clave en el documento de requerimientos. Después relacionamos cada una de ellas con las clases específicas en nuestro sistema (figura 6.20). Las frases verbales en la figura 6.20 nos ayudan a determinar las operaciones de cada clase.

Modelar las operaciones

Para identificar las operaciones, analizamos las frases verbales que se listan para cada clase en la figura 6.20. La frase "ejecuta transacciones financieras" asociada con la clase `ATM` implica que esta clase instruye a las transacciones a que se ejecuten. Por lo tanto, cada una de las clases `SolicitudSaldo`, `Retiro` y `Deposito` necesitan una operación para proporcionar este servicio al ATM. Colocamos esta operación (que hemos nombrado `ejecutar`) en

Clase	Verbos y frases verbales
`ATM`	ejecuta transacciones financieras
`SolicitudSaldo`	[ninguna en el documento de requerimientos]
`Retiro`	[ninguna en el documento de requerimientos]
`Deposito`	[ninguna en el documento de requerimientos]
`BaseDatosBanco`	autentica a un usuario, obtiene el saldo de una cuenta, abona un monto de depósito a una cuenta, carga un monto de retiro a una cuenta
`Cuenta`	obtiene el saldo de una cuenta, abona un monto de depósito a una cuenta, carga un monto de retiro a una cuenta
`Pantalla`	muestra un mensaje al usuario
`Teclado`	recibe entrada numérica del usuario
`DispensadorEfectivo`	dispensa efectivo, indica si contiene suficiente efectivo para satisfacer una solicitud de retiro
`RanuraDeposito`	recibe un sobre de depósito

Figura 6.20 | Verbos y frases verbales para cada clase en el sistema ATM.

el tercer compartimiento de las tres clases de transacciones en el diagrama de clases actualizado de la figura 6.21. Durante una sesión con el ATM, el objeto `ATM` invocará estas operaciones de transacciones, según sea necesario.

Para representar las operaciones (que se implementan en forma de métodos en Java), UML lista el nombre de la operación, seguido de una lista separada por comas de parámetros entre paréntesis, un signo de punto y coma y el tipo de valor de retorno:

nombreOperación(*parámetro1*, *parámetro2*, …, *parámetroN*) : *tipo de valor de retorno*

Cada parámetro en la lista separada por comas consiste en un nombre de parámetro, seguido de un signo de dos puntos y del tipo del parámetro:

nombreParámetro : *tipoParámetro*

Por el momento, no listamos los parámetros de nuestras operaciones; en breve identificaremos y modelaremos los parámetros de algunas de las operaciones. Para algunas de estas operaciones no conocemos todavía los tipos de valores de retorno, por lo que también las omitiremos del diagrama. Estas omisiones son perfectamente normales en este punto. A medida que avancemos en nuestro proceso de diseño e implementación, agregaremos el resto de los tipos de valores de retorno.

La figura 6.20 lista la frase "autentica a un usuario" enseguida de la clase `BaseDatosBanco`; la base de datos es el objeto que contiene la información necesaria de la cuenta para determinar si el número de cuenta y el NIP introducidos por un usuario concuerdan con los de una cuenta en el banco. Por lo tanto, la clase `BaseDatos-Banco` necesita una operación que proporcione un servicio de autenticación al ATM. Colocamos la operación

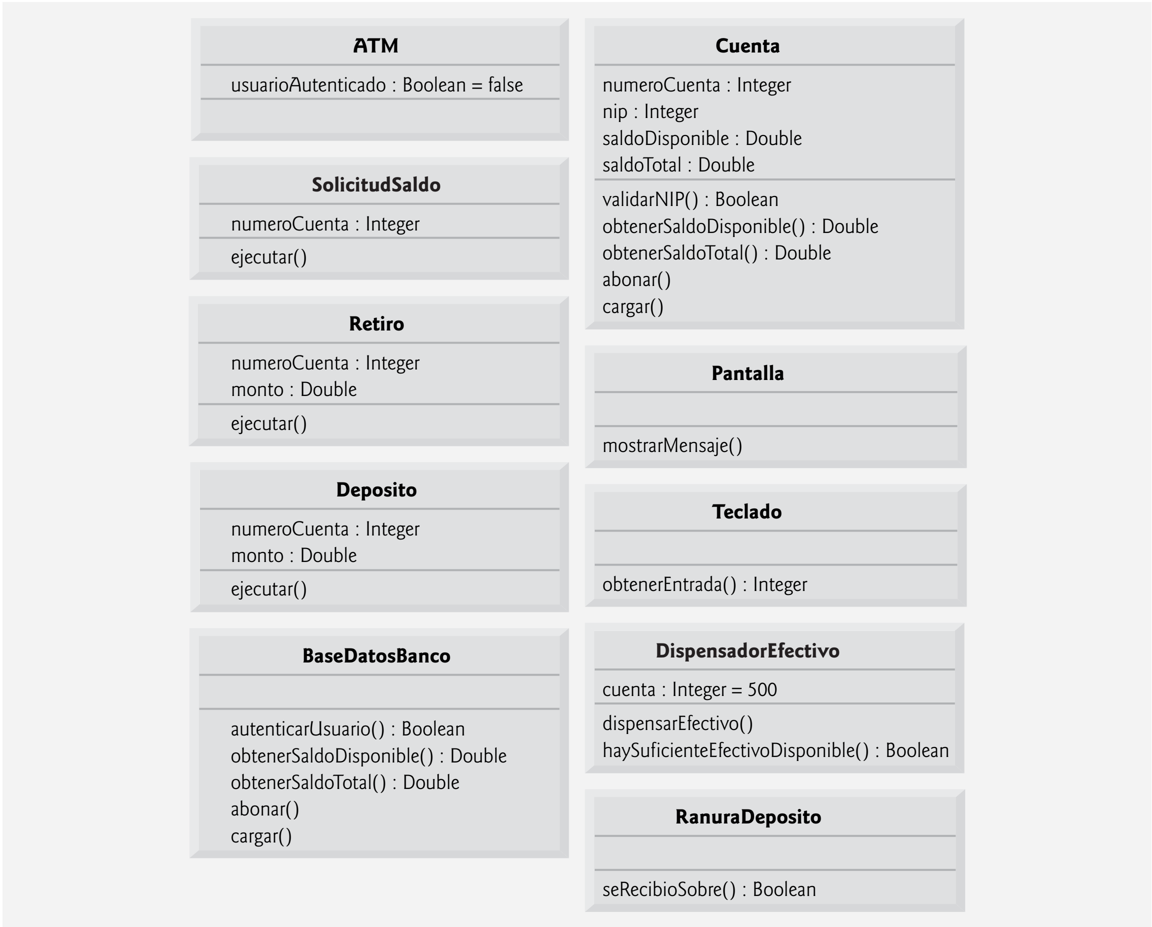


Figura 6.21 | Las clases en el sistema ATM, con atributos y operaciones.

autenticarUsuario en el tercer compartimiento de la clase BaseDatosBanco (figura 6.21). No obstante, un objeto de la clase Cuenta y no de la clase BaseDatosBanco es el que almacena el número de cuenta y el NIP a los que se debe acceder para autenticar a un usuario, por lo que la clase Cuenta debe proporcionar un servicio para validar un NIP obtenido como entrada del usuario, y compararlo con un NIP almacenado en un objeto Cuenta. Por ende, agregamos una operación validarNIP a la clase Cuenta. Observe que especificamos un tipo de valor de retorno Boolean para las operaciones autenticarUsuario y validarNIP. Cada operación devuelve un valor que indica que la operación tuvo éxito al realizar su tarea (es decir, un valor de retorno true) o que no tuvo éxito (es decir, un valor de retorno false).

La figura 6.20 lista varias frases verbales adicionales para la clase BaseDatosBanco: "extrae el saldo de una cuenta", "abona un monto de depósito a una cuenta" y "carga un monto de retiro a una cuenta". Al igual que "autentica a un usuario", estas frases restantes se refieren a los servicios que debe proporcionar la base de datos al ATM, ya que la base de datos almacena todos los datos de las cuentas que se utilizan para autenticar a un usuario y realizar transacciones con el ATM. No obstante, los objetos de la clase Cuenta son los que en realidad realizan las operaciones a las que se refieren estas frases. Por ello, asignamos una operación tanto a la clase BaseDatosBanco como a la clase Cuenta, que corresponda con cada una de estas frases. En la sección 3.10 vimos que, como una cuenta de banco contiene información delicada, no permitimos que el ATM acceda a las cuentas en forma directa. La base de datos actúa como un intermediario entre el ATM y los datos de la cuenta, evitando el acceso no autorizado. Como veremos en la sección 7.14, la clase ATM invoca las operaciones de la clase BaseDatosBanco, cada una de las cuales a su vez invoca a la operación con el mismo nombre en la clase Cuenta.

La frase "obtiene el saldo de una cuenta" sugiere que las clases BaseDatosBanco y Cuenta necesitan una operación obtenerSaldo. Sin embargo, recuerde que creamos dos atributos en la clase Cuenta para representar un saldo: saldoDisponible y saldoTotal. Una solicitud de saldo requiere el acceso a estos dos atributos del saldo, de manera que pueda mostrarlos al usuario, pero un retiro sólo requiere verificar el valor de saldoDisponible. Para permitir que los objetos en el sistema obtengan cada atributo de saldo en forma individual, agregamos las operaciones obtenerSaldoDisponible y obtenerSaldoTotal al tercer compartimiento de las clases BaseDatosBanco y Cuenta (figura 6.21). Especificamos un tipo de valor de retorno Double para estas operaciones, debido a que los atributos de los saldos que van a obtener son de tipo Double.

Las frases "abona un monto de depósito a una cuenta" y "carga un monto de retiro a una cuenta" indican que las clases BaseDatosBanco y Cuenta deben realizar operaciones para actualizar una cuenta durante un depósito y un retiro, respectivamente. Por lo tanto, asignamos las operaciones abonar y cargar a las clases BaseDatosBanco y Cuenta. Tal vez recuerde que cuando se abona a una cuenta (como en un depósito) se suma un monto sólo al atributo saldoTotal. Por otro lado, cuando se carga a una cuenta (como en un retiro) se resta el monto tanto del saldo total como del saldo disponible. Ocultamos estos detalles de implementación dentro de la clase Cuenta. Éste es un buen ejemplo de encapsulamiento y ocultamiento de información.

Si éste fuera un sistema ATM real, las clases BaseDatosBanco y Cuenta también proporcionarían un conjunto de operaciones para permitir que otro sistema bancario actualizara el saldo de la cuenta de un usuario después de confirmar o rechazar todo, o parte de, un depósito. Por ejemplo, la operación confirmarMontoDeposito sumaría un monto al atributo saldoDisponible, y haría que los fondos depositados estuvieran disponibles para retirarlos. La operación rechazarMontoDeposito restaría un monto al atributo saldoTotal para indicar que un monto especificado, que se había depositado recientemente a través del ATM y se había sumado al saldoTotal, no se encontró en el sobre de depósito. El banco invocaría esta operación después de determinar que el usuario no incluyó el monto correcto de efectivo o que algún cheque no fue validado (es decir, que "rebotó"). Aunque al agregar estas operaciones nuestro sistema estaría más completo, no las incluiremos en nuestros diagramas de clases ni en nuestra implementación, ya que se encuentran más allá del alcance de este ejemplo práctico.

La clase Pantalla "muestra un mensaje al usuario" en diversos momentos durante una sesión con el ATM. Toda la salida visual se produce a través de la pantalla del ATM. El documento de requerimientos describe muchos tipos de mensajes (por ejemplo, un mensaje de bienvenida, un mensaje de error, un mensaje de agradecimiento) que la pantalla muestra al usuario. El documento de requerimientos también indica que la pantalla muestra indicadores y menús al usuario. No obstante, un indicador es en realidad sólo un mensaje que describe lo que el usuario debe introducir a continuación, y un menú es en esencia un tipo de indicador que consiste en una serie de mensajes (es decir, las opciones del menú) que se muestran en forma consecutiva. Por lo tanto, en vez de asignar a la clase Pantalla una operación individual para mostrar cada tipo de mensaje, indicador y menú, basta con crear una operación que pueda mostrar cualquier mensaje especificado por un parámetro. Colocamos esta operación (mostrarMensaje) en el tercer compartimiento de la clase Pantalla en nuestro diagrama de

clases (figura 6.21). Observe que no nos preocupa el parámetro de esta operación en estos momentos; lo modelaremos más adelante en esta sección.

De la frase "recibe entrada numérica del usuario" listada por la clase `Teclado` en la figura 6.20, podemos concluir que la clase `Teclado` debe realizar una operación `obtenerEntrada`. A diferencia del teclado de una computadora, el teclado del ATM sólo contiene los números del 0 al 9, por lo cual especificamos que esta operación devuelve un valor entero. Si recuerda, en el documento de requerimientos vimos que en distintas situaciones, tal vez se requiera que el usuario introduzca un tipo distinto de número (por ejemplo, un número de cuenta, un NIP, el número de una opción del menú, un monto de depósito como número de centavos). La clase `Teclado` sólo obtiene un valor numérico para un cliente de la clase; no determina si el valor cumple con algún criterio específico. Cualquier clase que utilice esta operación debe verificar que el usuario haya introducido un número apropiado según el caso, y después debe responder de manera acorde (por ejemplo, mostrar un mensaje de error a través de la clase `Pantalla`). [*Nota:* cuando implementemos el sistema, simularemos el teclado del ATM con el teclado de una computadora y, por cuestión de simpleza, asumiremos que el usuario no escribirá datos de entrada que no sean números, usando las teclas en el teclado de la computadora que no aparezcan en el teclado del ATM].

La figura 6.20 lista la frase "dispensa efectivo" para la clase `DispensadorEfectivo`. Por lo tanto, creamos la operación `dispensarEfectivo` y la listamos bajo la clase `DispensadorEfectivo` en la figura 6.21. La clase `DispensadorEfectivo` también "indica si contiene suficiente efectivo para satisfacer una solicitud de retiro". Para esto incluimos a `haySuficienteEfectivoDisponible`, una operación que devuelve un valor de tipo `Boolean` de UML, en la clase `DispensadorEfectivo`. La figura 6.20 también lista la frase "recibe un sobre de depósito" para la clase `RanuraDeposito`. La ranura de depósito debe indicar si recibió un sobre, por lo que colocamos una operación `seRecibioSobre`, la cual devuelve un valor `Boolean`, en el tercer compartimiento de la clase `RanuraDeposito`. [*Nota:* es muy probable que una ranura de depósito de hardware real envíe una señal al ATM para indicarle que se recibió un sobre. No obstante, simularemos este comportamiento con una operación en la clase `RanuraDeposito`, que la clase `ATM` pueda invocar para averiguar si la ranura de depósito recibió un sobre].

No listamos ninguna operación para la clase `ATM` en este momento. Todavía no sabemos de algún servicio que proporcione la clase `ATM` a otras clases en el sistema. No obstante, cuando implementemos el sistema en código de Java, tal vez emerjan las operaciones de esta clase junto con las operaciones adicionales de las demás clases en el sistema.

Identificar y modelar los parámetros de operación

Hasta ahora no nos hemos preocupado por los parámetros de nuestras operaciones; sólo hemos tratado de obtener una comprensión básica de las operaciones de cada clase. Ahora daremos un vistazo más de cerca a varios parámetros de operación. Para identificar los parámetros de una operación, analizamos qué datos requiere la operación para realizar su tarea asignada.

Considere la operación `autenticarUsuario` de la clase `BaseDatosBanco`. Para autenticar a un usuario, esta operación debe conocer el número de cuenta y el NIP que suministra el usuario. Por lo tanto, especificamos que la operación `autenticarUsuario` debe recibir los parámetros enteros `numeroCuentaUsuario` y `nipUsuario`, que la operación debe comparar con el número de cuenta y el NIP de un objeto `Cuenta` en la base de datos. Colocaremos después de estos nombres de parámetros la palabra `Usuario`, para evitar confusión entre los nombres de los parámetros de la operación y los nombres de los atributos que pertenecen a la clase `Cuenta`. Listamos estos parámetros en el diagrama de clases de la figura 6.22, el cual modela sólo a la clase `BaseDatosBanco`. [*Nota:* es perfectamente normal modelar sólo una clase en un diagrama de clases. En este caso lo que más nos preocupa es analizar los parámetros de esta clase específica, por lo que omitimos las demás clases. Más adelante en los diagramas de clase de este ejemplo práctico, en donde los parámetros dejarán de ser el centro de nuestra atención, los omitiremos para ahorrar espacio. No obstante, recuerde que las operaciones que se listan en estos diagramas siguen teniendo parámetros].

Recuerde que para modelar a cada parámetro en una lista de parámetros separados por comas, UML lista el nombre del parámetro, seguido de un signo de dos puntos y el tipo del parámetro (en notación de UML). Así, la figura 6.22 especifica que la operación `autenticarUsuario` recibe dos parámetros: `numeroCuentaUsuario` y `nipUsuario`, ambos de tipo `Integer`. Cuando implementemos el sistema en Java, representaremos estos parámetros con valores `int`.

Las operaciones `obtenerSaldoDisponible`, `obtenerSaldoTotal`, `abonar` y `cargar` de la clase `BaseDatosBanco` también requieren un parámetro `nombreCuentaUsuario` para identificar la cuenta a la cual la base de datos debe aplicar las operaciones, por lo que incluimos estos parámetros en el diagrama de clases de la figura

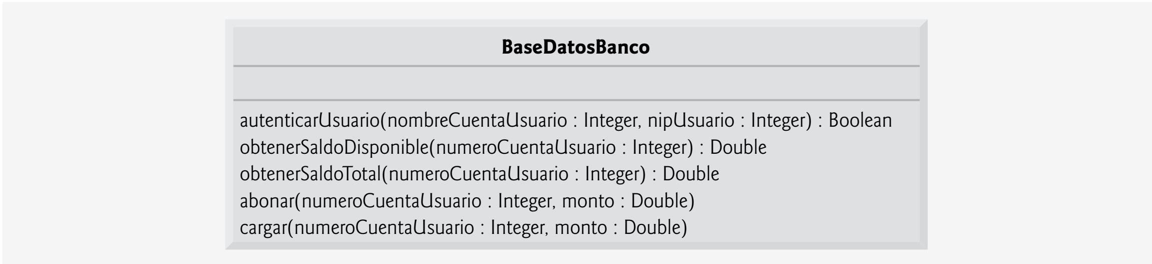


figura 6.22 | La clase `BaseDatosBanco` con parámetros de operación.

6.22. Además, las operaciones `abonar` y `cargar` requieren un parámetro `Double` llamado `monto`, para especificar el monto de dinero que se abonará o cargará, respectivamente.

El diagrama de clases de la figura 6.23 modela los parámetros de las operaciones de la clase `Cuenta`. La operación `validarNIP` sólo requiere un parámetro `nipUsuario`, el cual contiene el NIP especificado por el usuario, que se comparará con el NIP asociado a la cuenta. Al igual que sus contrapartes en la clase `BaseDatosBanco`, las operaciones `abonar` y `cargar` en la clase `Cuenta` requieren un parámetro `Double` llamado `monto`, el cual indica la cantidad de dinero involucrada en la operación. Las operaciones `obtenerSaldoDisponible` y `obtenerSaldoTotal` en la clase `Cuenta` no requieren datos adicionales para realizar sus tareas. Observe que las operaciones de la clase `Cuenta` no requieren un parámetro de número de cuenta para diferenciar una cuenta de otra, ya que cada una de estas operaciones se puede invocar sólo en un objeto `Cuenta` específico.

La figura 6.24 modela la clase `Pantalla` con un parámetro especificado para la operación `mostrarMensaje`. Esta operación requiere sólo un parámetro `String` llamado `mensaje`, el cual indica el texto que debe mostrarse en pantalla. Recuerde que los tipos de los parámetros que se enlistan en nuestros diagramas de clases están en notación de UML, por lo que el tipo `String` que se enlista en la figura 6.24 se refiere al tipo de UML. Cuando implementemos el sistema en Java, utilizaremos de hecho la clase `String` de Java para representar este parámetro.

El diagrama de clases de la figura 6.25 especifica que la operación `dispensarEfectivo` de la clase `DispensadorEfectivo` recibe un parámetro `Double` llamado `monto` para indicar el monto de efectivo (en dólares) que se dispensará al usuario. La operación `haySuficienteEfectivoDisponible` también recibe un parámetro `Double` llamado `monto` para indicar el monto de efectivo en cuestión.

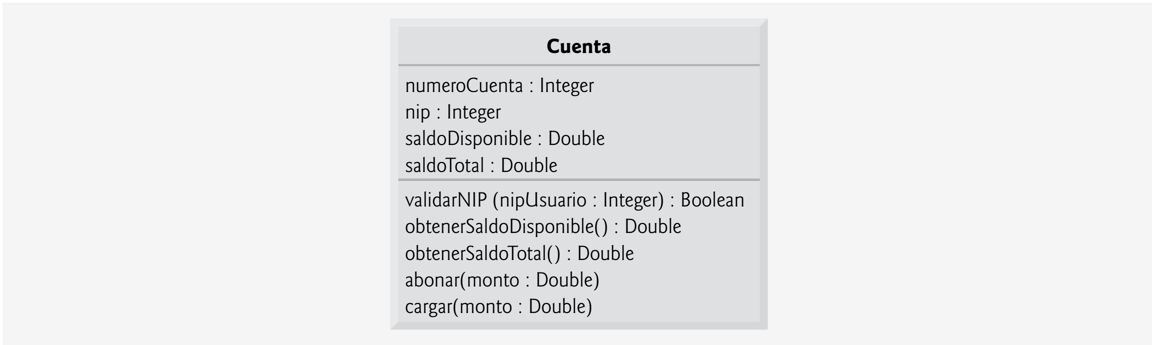


Figura 6.23 | La clase `Cuenta` con parámetros de operación.

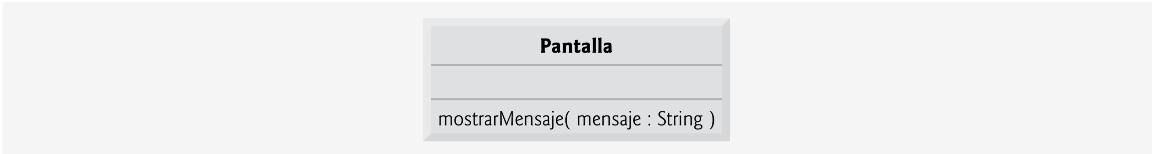


Figura 6.24 | La clase `Pantalla` con parámetros de operación.

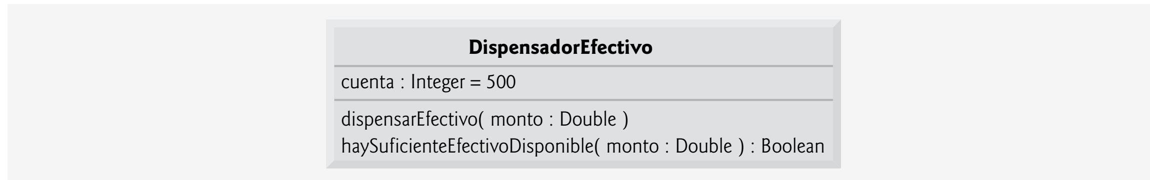


Figura 6.25 | La clase `DispensadorEfectivo` con parámetros de operación.

Observe que no hablamos sobre los parámetros para la operación `ejecutar` de las clases `SolicitudSaldo`, `Retiro` y `Depósito`, de la operación `obtenerEntrada` de la clase `Teclado` y la operación `seRecibioSobre` de la clase `RanuraDeposito`. En este punto de nuestro proceso de diseño, no podemos determinar si estas operaciones requieren datos adicionales para realizar sus tareas, por lo que dejaremos sus listas de parámetros vacías. A medida que avancemos por el ejemplo práctico, tal vez decidamos agregar parámetros a estas operaciones.

En esta sección hemos determinado muchas de las operaciones que realizan las clases en el sistema ATM. Identificamos los parámetros y los tipos de valores de retorno de algunas operaciones. A medida que continuemos con nuestro proceso de diseño, el número de operaciones que pertenezcan a cada clase puede variar; podríamos descubrir que se necesitan nuevas operaciones o que ciertas operaciones actuales no son necesarias; y podríamos determinar que algunas de las operaciones de nuestras clases necesitan parámetros adicionales y tipos de valores de retorno distintos.

Ejercicios de autoevaluación del Ejemplo práctico de Ingeniería de Software

6.1 ¿Cuál de las siguientes opciones no es un comportamiento?
a) Leer datos de un archivo.
b) Imprimir los resultados.
c) Imprimir texto.
d) Obtener la entrada del usuario.

6.2 Si quisiera agregar al sistema ATM una operación que devuelva el atributo `monto` de la clase `Retiro`, ¿cómo y en dónde especificaría esta operación en el diagrama de clases de la figura 6.21?

6.3 Describa el significado del siguiente listado de operaciones, el cual podría aparecer en un diagrama de clases para el diseño orientado a objetos de una calculadora:

```
sumar( x : Integer, y : Integer ) : Integer
```

Respuestas a los ejercicios de autoevaluación del Ejemplo práctico de Ingeniería de Software

6.1 c.

6.2 Para especificar una operación que obtenga el atributo `monto` de la clase `Retiro`, se debe colocar el siguiente listado de operaciones en el (tercer) compartimiento de operaciones de la clase `Retiro`:

```
obtenerMonto( ) : Double
```

6.3 Este listado de operaciones indica una operación llamada `sumar`, la cual recibe los enteros `x` y `y` como parámetros y devuelve un valor entero.

6.15 Conclusión

En este capítulo aprendió más acerca de los detalles de la declaración de métodos. También conoció la diferencia entre los métodos `static` y los no `static`, y le mostramos cómo llamar a los métodos `static`, anteponiendo al nombre del método el nombre de la clase en la cual aparece, y el separador punto (`.`). Aprendió a utilizar el operador `+` para realizar concatenaciones de cadenas. Aprendió a declarar constantes con nombre, usando los tipos `enum` y las variables `public final static`. Vio cómo usar la clase `Random` para generar conjuntos de números aleatorios, que pueden usarse para simulaciones. También aprendió acerca del alcance de los campos y las variables locales en una clase. Por último, aprendió que varios métodos en una clase pueden sobrecargarse, al proporcionar métodos con el mismo nombre y distintas firmas. Dichos métodos pueden usarse para realizar las mismas tareas, o tareas similares, usando distintos tipos o distintos números de parámetros.

En el capítulo 7 aprenderá a mantener listas y tablas de datos en arreglos. Verá una implementación más elegante de la aplicación que tira un dado 6000 veces, y dos versiones mejoradas de nuestro ejemplo práctico `LibroCalificaciones` que estudió en los capítulos 3 a 5. También aprenderá cómo acceder a los argumentos de línea de comandos de una aplicación, los cuales se pasan al método `main` cuando una aplicación comienza su ejecución.

Resumen

Sección 6.1 Introducción

- La experiencia ha demostrado que la mejor forma de desarrollar y mantener un programa extenso es construirlo a partir de piezas pequeñas y simples, o módulos. A esta técnica se le conoce como "divide y vencerás".

Sección 6.2 Módulos de programas en Java

- Hay tres tipos de módulos en Java: métodos, clases y paquetes. Los métodos se declaran dentro de las clases. Por lo general, las clases se agrupan en paquetes para que puedan importarse en los programas y reutilizarse.
- Los métodos nos permiten dividir un programa en módulos, al separar sus tareas en unidades autocontenidas. Las instrucciones en un método se escriben sólo una vez, y se ocultan de los demás métodos.
- Utilizar los métodos existentes como bloques de construcción para crear nuevos programas es una forma de reutilización del software, que nos permite evitar repetir código dentro de un programa.

Sección 6.3 Métodos `static`, campos `static` y la clase `Math`

- Una llamada a un método especifica el nombre del método a llamar y proporciona los argumentos que el método al que se llamó requiere para realizar su tarea. Cuando termina la llamada al método, éste devuelve un resultado o simplemente devuelve el control al método que lo llamó.
- Una clase puede contener métodos static para realizar tareas comunes que no requieren un objeto de la clase. Cualquier información que pueda requerir un método static para realizar sus tareas se le puede enviar en forma de argumentos, en una llamada al método. Para llamar a un método static, se especifica el nombre de la clase en la cual está declarado el método, seguido de un punto (`.`) y del nombre del método, como en

  NombreClase.nombreMétodo`(` *argumentos* `)`

- Los argumentos para los métodos pueden ser constantes, variables o expresiones.
- La clase `Math` cuenta con métodos `static` para realizar cálculos matemáticos comunes; además, declara dos campos que representan constantes matemáticas de uso común: `Math.PI` y `Math.E`. La constante `Math.PI` (3.14159265358979323846) es la relación entre la circunferencia de un círculo y su diámetro. La constante `Math.E` (2.7182818284590452354) es el valor de la base para los logaritmos naturales (que se calculan con el método `static Math log`).
- `Math.PI` y `Math.E` se declaran con los modificadores `public`, `final` y `static`. Al hacerlos `public`, otros programadores pueden usar estos campos en sus propias clases. Cualquier campo declarado con la palabra clave `final` es constante; su valor no se puede modificar una vez que se inicializa el campo. Tanto `PI` como `E` se declaran `final`, ya que sus valores nunca cambian. Al hacer a estos campos `static`, se puede acceder a ellos a través del nombre de la clase `Math` y un separador punto (`.`), justo igual que con los métodos de la clase `Math`.
- Cuando se crean objetos de una clase que contiene campos `static` (variables de clase), todos los objetos de esa clase comparten una copia de los campos `static`. En conjunto, las variables de clase y las variables de instancia de la clase representan sus campos. En la sección 8.11 aprenderá más acerca de los campos `static`.
- Al ejecutar la Máquina Virtual de Java (JVM) con el comando `java`, la JVM trata de invocar al método `main` de la clase que usted le especifique. La JVM carga la clase especificada por *NombreClase* y utiliza el nombre de esa clase para invocar al método `main`. Puede especificar una lista opcional de objetos `String` (separados por espacios) como argumentos de línea de comandos, que la JVM pasará a su aplicación.
- Puede colocar un método `main` en cualquier clase que declare; sólo se llamará al método `main` en la clase que usted utilice para ejecutar la aplicación. Algunos programadores aprovechan esto para crear un pequeño programa de prueba en cada clase que declaran.

Sección 6.4 Declaración de métodos con múltiples parámetros

- Cuando se hace una llamada a un método, el programa crea una copia de los valores de los argumentos del método y los asigna a los parámetros correspondientes del mismo, que se crean e inicializan cuando se hace la llamada al método. Cuando el control del programa regresa al punto en el que se hizo la llamada al método, los parámetros del mismo se eliminan de la memoria.
- Un método puede devolver a lo más un valor, pero el valor devuelto podría ser una referencia a un objeto que contenga muchos valores.
- Las variables deben declararse como campos de una clase, sólo si se requieren para usarlos en más de un método de la clase, o si el programa debe guardar sus valores entre distintas llamadas a los métodos de la clase.

Sección 6.5 Notas acerca de cómo declarar y utilizar los métodos

- Hay tres formas de llamar a un método: usar el nombre de un método por sí solo para llamar a otro método de la misma clase; usar una variable que contenga una referencia a un objeto, seguida de un punto (`.`) y del nombre del método, para llamar a un método del objeto al que se hace referencia; y usar el nombre de la clase y un punto (`.`) para llamar a un método `static` de una clase.
- Hay tres formas de devolver el control a una instrucción que llama a un método. Si el método no devuelve un resultado, el control regresa cuando el flujo del programa llega a la llave derecha de terminación del método, o cuando se ejecuta la instrucción

  ```
  return;
  ```

  si el método devuelve un resultado, la instrucción

  `return` *expresión*;

  evalúa la *expresión*, y después regresa de inmediato el valor resultante al método que hizo la llamada.
- Cuando un método tiene más de un parámetro, los parámetros se especifican como una lista separada por comas. Debe haber un argumento en la llamada al método para cada parámetro en su declaración. Además, cada argumento debe ser consistente con el tipo del parámetro correspondiente. Si un método no acepta argumentos, la lista de parámetros está vacía.
- Los objetos `String` se pueden concatenar mediante el uso del operador `+`, que coloca los caracteres del operando derecho al final de los que están en el operando izquierdo.
- Cada valor primitivo y objeto en Java tiene una representación `String`. Cuando se concatena un objeto con un `String`, el objeto se convierte en un `String` y después, los dos `String` se concatenan.
- Para los valores primitivos que se utilizan en la concatenación de cadenas, la JVM maneja la conversión de los valores primitivos a objetos `String`. Si un valor `boolean` se concatena con un objeto `String`, se utiliza la palabra `"true"` o la palabra `"false"` para representar el valor `boolean`. Si hay ceros a la derecha en un valor de punto flotante, se descartan cuando el número se concatena a un objeto `String`.
- Todos los objetos en Java tienen un método especial, llamado `toString`, el cual devuelve una representación `String` del contenido del objeto. Cuando se concatena un objeto con un `String`, la JVM llama de manera implícita al método `toString` del objeto, para obtener la representación `String` del mismo.
- Cuando se escribe una literal `String` extensa en el código fuente de un programa, algunas veces los programadores dividen esa literal `String` en varias literales `String` más pequeñas, y las colocan en varias líneas de código para mejorar la legibilidad, y después vuelven a ensamblar las literales `String` mediante la concatenación.

Sección 6.6 Pila de llamadas a los métodos y registros de activación

- Las pilas se conocen como estructuras de datos tipo "último en entrar, primero en salir (UEPS)"; el último elemento que se mete (inserta) en la pila es el primer elemento que se saca (extrae) de ella.
- Un método al que se llama debe saber cómo regresar al método que lo llamó, por lo que la dirección de retorno del método que hace la llamada se mete en la pila de ejecución del programa cuando se llama al método. Si ocurre una serie de llamadas a métodos, las direcciones de retorno sucesivas se meten en la pila, en el orden último en entrar, primero en salir, de manera que el último método en ejecutarse sea el primero en regresar al método que lo llamó.
- La pila de ejecución del programa contiene la memoria para las variables locales que se utilizan en cada invocación de un método, durante la ejecución de un programa. Este dato se conoce como el registro de activación, o marco de pila, de la llamada al método. Cuando se hace una llamada a un método, el registro de activación para la llamada a ese método se mete en la pila de ejecución del programa. Cuando el método regresa al método que lo llamó, el registro de activación para esta llamada al método se saca de la pila, y esas variables locales ya no son conocidas para el programa. Si una variable local que contiene una referencia a un objeto es la única variable en el programa con una

referencia a ese objeto, cuando el registro de activación que contiene esa variable local se saca de la pila, el programa ya no puede acceder al objeto y, en un momento dado, la JVM lo eliminará de la memoria durante la "recolección de basura".

- La cantidad de memoria en una computadora es finita, por lo que sólo puede utilizarse cierta cantidad de memoria para almacenar registros de activación en la pila de ejecución del programa. Si hay más llamadas a métodos de las que se puedan almacenar en sus registros de activación en la pila de ejecución del programa, se produce un error conocido como desbordamiento de pila. La aplicación se compilará correctamente, pero su ejecución producirá un desbordamiento de pila.

Sección 6.7 Promoción y conversión de argumentos

- Una característica importante de las llamadas a métodos es la promoción de argumentos: convertir el valor de un argumento al tipo que el método espera recibir en su parámetro correspondiente.
- Hay un conjunto de reglas de promoción que se aplican a las expresiones que contienen valores de dos o más tipos primitivos, y a los valores de tipos primitivos que se pasan como argumentos para los métodos. Cada valor se promueve al tipo "más alto" en la expresión. En casos en los que se puede perder información debido a la conversión, el compilador de Java requiere que utilicemos un operador de conversión de tipos para obligar explícitamente a que ocurra la conversión.

Sección 6.9 Ejemplo práctico: generación de números aleatorios

- Los objetos de la clase `Random` (paquete `java.util`) pueden producir valores `int`, `long`, `float` o `double`. El método `random` de `Math` puede producir valores `double` en el rango $0.0 \le x < 1.0$, en donde x es el valor devuelto por el método `random`.
- El método `nextInt` de `Random` genera un valor `int` aleatorio en el rango de –2,147,483,648 a +2,147,483,647. Los valores devueltos por `nextInt` son en realidad números seudoaleatorios: una secuencia de valores producidos por un cálculo matemático complejo. Ese cálculo utiliza la hora actual del día para sembrar el generador de números aleatorios, de tal forma que cada ejecución del programa produzca una secuencia diferente de valores aleatorios.
- La clase `Random` cuenta con otra versión del método `nextInt`, la cual recibe un argumento `int` y devuelve un valor desde 0 hasta el valor del argumento (pero sin incluirlo).
- Los números aleatorios en un rango pueden generarse mediante

```
numero = valorDesplazamiento + numerosAleatorios.nextInt( factorEscala );
```

  en donde *valorDesplazamiento* especifica el primer número en el rango deseado de enteros consecutivos, y *factorEscala* especifica cuántos números hay en el rango.
- Los números aleatorios pueden elegirse a partir de rangos de enteros no consecutivos, como en

```
numero = valorDesplazamiento +
    diferenciaEntreValores * numerosAleatorios.nextInt( factorEscala );
```

  en donde *valorDesplazamiento* especifica el primer número en el rango de valores, *diferenciaEntreValores* representa la diferencia entre números consecutivos en la secuencia y *factorEscala* especifica cuántos números hay en el rango.
- Para depurar, algunas veces es conveniente repetir la misma secuencia de números seudoaleatorios durante cada ejecución del programa, para demostrar que su aplicación funciona para una secuencia específica de números aleatorios, antes de probar el programa con distintas secuencias de números aleatorios. Cuando la repetitividad es importante, puede crear un objeto `Random` al pasar un valor entero `long` al constructor. Si se utiliza la misma semilla cada vez que se ejecuta el programa, el objeto `Random` produce la misma secuencia de números aleatorios. También puede establecer la semilla de un objeto `Random` en cualquier momento, llamando al método `setSeed` del objeto.

Sección 6.10 Ejemplo práctico: un juego de probabilidad (introducción a las enumeraciones)

- Una enumeración se introduce mediante la palabra clave `enum` y el nombre de un tipo. Al igual que con cualquier clase, las llaves (`{` y `}`) delimitan el cuerpo de una declaración `enum`. Dentro de las llaves hay una lista separada por comas de constantes de enumeración, cada una de las cuales representa un valor único. Los identificadores en una `enum` deben ser únicos. A las variables de tipo `enum` sólo se les pueden asignar constantes de ese tipo `enum`.
- Las constantes también pueden declararse como variables `public final static`. Dichas constantes se declaran todas con letras mayúsculas por convención, para hacer que resalten en el programa.

Sección 6.11 Alcance de las declaraciones

- El alcance es la porción del programa en la que se puede hacer referencia a una entidad, como una variable o un método, por su nombre. Se dice que dicha entidad está "dentro del alcance" para esa porción del programa.

- El alcance de la declaración de un parámetro es el cuerpo del método en el que aparece esa declaración.
- El alcance de la declaración de una variable local es a partir del punto en el que aparece la declaración, hasta el final de ese bloque.
- El alcance de una etiqueta en una instrucción `break` o `continue` etiquetada es el cuerpo de la instrucción etiquetada.
- El alcance de la declaración de una variable local que aparece en la sección de inicialización del encabezado de una instrucción `for` es el cuerpo de la instrucción `for`, junto con las demás expresiones en el encabezado.
- El alcance de un método o campo de una clase es todo el cuerpo de la clase. Esto permite que los métodos de una clase utilicen nombres simples para llamar a los demás métodos de la clase y acceder a los campos de la misma.
- Cualquier bloque puede contener declaraciones de variables. Si una variable local o parámetro en un método tiene el mismo nombre que un campo, éste se oculta hasta que el bloque termina de ejecutarse.

Sección 6.12 Sobrecarga de métodos

- Java permite que se declaren varios métodos con el mismo nombre en una clase, siempre y cuando los métodos tengan distintos conjuntos de parámetros (lo cual se determina en base al número, orden y tipos de los parámetros). A esta técnica se le conoce como sobrecarga de métodos.
- Los métodos sobrecargados se distinguen por sus firmas: combinaciones de los nombres de los métodos y el número, tipos y orden de sus parámetros. Los métodos no pueden distinguirse en base al tipo de valor de retorno.

Terminología

alcance de una declaración
argumento de línea de comandos
bloque
campos “ocultos”
`Color`, clase
componentes de software reutilizables
concatenación de cadenas
constante de enumeración
declaración de un método
desbordamiento de pila
desplazar un rango (números aleatorios)
dividir en módulos un programa con métodos
documentación de la API de Java
elemento de probabilidad
`enum`, palabra clave
enumeración
extraer (de una pila)
factor de escala (números aleatorios)
`fillOval`, método de la clase `Graphics`
`fillRect`, método de la clase `Graphics`
`final`, palabra clave
firma de un método
función
insertar (en una pila)
interfaz de programación de aplicaciones (API)
Interfaz de programación de aplicaciones de Java (API)
invocar a un método
lista de parámetros
lista de parámetros separados por comas
llamada a método
marco de pila
método “divide y vencerás”
método de clase
método declarado por el programador
módulo
`nextInt`, método de la clase `Random`
número seudoaleatorio
números aleatorios
ocultar los detalles de implementación
ocultar un campo
paquete
parámetro
parámetro formal
pila
pila de ejecución del programa
pila de llamadas a métodos
procedimiento
promoción de argumentos
promociones de tipos primitivos
`random` de la clase `Math`
`Random`, clase
registro de activación
reglas de promoción
relación jerárquica método jefe/método trabajador
`return`, palabra clave
reutilización de software
`setColor`, método de la clase `Graphics`
`setSeed`, método de la clase `Random`
simulación
sobrecarga de métodos
sobrecargar un método
último en entrar, primero en salir (UEPS), estructura de datos
valor de desplazamiento (números aleatorios)
valor de semilla (números aleatorios)
valores RGB
variable de clase
variable local

Ejercicios de autoevaluación

6.1 Complete las siguientes oraciones:

a) Un método se invoca con un __________.
b) A una variable que se conoce sólo dentro del método en el que está declarada, se le llama __________.
c) La instrucción __________ en un método llamado puede usarse para regresar el valor de una expresión, al método que hizo la llamada.
d) La palabra clave __________ indica que un método no devuelve ningún valor.
e) Los datos pueden agregarse o eliminarse sólo desde __________ de una pila.
f) Las pilas se conocen como estructuras de datos__________: el último elemento que se mete (inserta) en la pila es el primer elemento que se saca (extrae) de ella.
g) Las tres formas de regresar el control de un método llamado a un solicitante son __________, __________ y __________.
h) Un objeto de la clase __________ produce números aleatorios.
i) La pila de ejecución del programa contiene la memoria para las variables locales en cada invocación de un método, durante la ejecución de un programa. Estos datos, almacenados como una parte de la pila de ejecución del programa, se conocen como __________ o __________ de la llamada al método.
j) Si hay más llamadas a métodos de las que puedan almacenarse en la pila de ejecución del programa, se produce un error conocido como __________.
k) El __________ de una declaración es la porción del programa que puede hacer referencia a la entidad en la declaración, por su nombre.
l) En Java, es posible tener varios métodos con el mismo nombre, en donde cada uno opere con distintos tipos o números de argumentos. A esta característica se le llama __________ de métodos.
m) La pila de ejecución del programa también se conoce como la pila de __________.

6.2 Para la clase `Craps` de la figura 6.9, indique el alcance de cada una de las siguientes entidades:

a) la variable `numerosAleatorios`.
b) la variable `dado1`.
c) el método `tirarDado`.
d) el método `jugar`.
e) la variable `sumaDeDados`.

6.3 Escriba una aplicación que pruebe si los ejemplos de las llamadas a los métodos de la clase `Math` que se muestran en la figura 6.2 realmente producen los resultados indicados.

6.4 Proporcione el encabezado para cada uno de los siguientes métodos:

a) El método `hipotenusa`, que toma dos argumentos de punto flotante con doble precisión, llamados `lado1` y `lado2`, y que devuelve un resultado de punto flotante, con doble precisión.
b) El método `menor`, que toma tres enteros `x`, `y` y `z`, y devuelve un entero.
c) El método `instrucciones`, que no toma argumentos y no devuelve ningún valor. (*Nota:* estos métodos se utilizan comúnmente para mostrar instrucciones a un usuario).
d) El método `intAFloat`, que toma un argumento entero llamado `numero` y devuelve un resultado de punto flotante.

6.5 Encuentre el error en cada uno de los siguientes segmentos de programas. Explique cómo se puede corregir el error.

a)
```
int g()
{
   System.out.println( "Dentro del metodo g" );
   int h()
   {
      System.out.println( "Dentro del método h" );
   }
}
```

b)
```
int suma( int x, int y )
{
     int resultado;
     resultado = x + y;
}
```
c)
```
voit f( float a );
{
     float a;
     System.out.println( a );
}
```
d)
```
voit producto()
{
     int a = 6, b = 5, c = 4, resultado;
     resultado = a * b * c;
     System.out.printf( "El resultado es %d\n", resultado );
     return resultado;
}
```

6.6 Escriba una aplicación completa en Java que pida al usuario el radio de tipo `double` de una esfera, y que llame al método `volumenEsfera` para calcular y mostrar el volumen de esa esfera. Utilice la siguiente asignación para calcular el volumen:

```
double volumen = ( 4.0 / 3.0 ) * Math.PI * Math.pow( radio, 3 )
```

Respuestas a los ejercicios de autoevaluación

6.1 a) llamada a un método. b) variable local. c) `return`. d) `void`. e) cima. f) último en entrar, primero en salir (UEPS). g) `return;` o `return` *expresión*; o encontrar la llave derecha de cierre de un método. h) `Random`. i) registro de activación. j) desbordamiento de pila. k) alcance. l) sobrecarga de métodos. m) llamadas a métodos.

6.2 a) el cuerpo de la clase. b) el bloque que define el cuerpo del método `tirarDado`. c) el cuerpo de la clase. d) el cuerpo de la clase. e) el bloque que define el cuerpo del método `jugar`.

6.3 La siguiente solución demuestra el uso de los métodos de la clase `Math` de la figura 6.2:

```
// Ejercicio 6.3: PruebaMath.java
// Prueba de los métodos de la clase Math.

public class PruebaMath
{
   public static void main( String args[] )
   {
      System.out.printf( "Math.abs( 23.7 ) = %f\n", Math.abs( 23.7 ) );
      System.out.printf( "Math.abs( 0.0 ) = %f\n", Math.abs( 0.0 ) );
      System.out.printf( "Math.abs( -23.7 ) = %f\n", Math.abs( -23.7 ) );
      System.out.printf( "Math.ceil( 9.2 ) = %f\n", Math.ceil( 9.2 ) );
      System.out.printf( "Math.ceil( -9.8 ) = %f\n", Math.ceil( -9.8 ) );
      System.out.printf( "Math.cos( 0.0 ) = %f\n", Math.cos( 0.0 ) );
      System.out.printf( "Math.exp( 1.0 ) = %f\n", Math.exp( 1.0 ) );
      System.out.printf( "Math.exp( 2.0 ) = %f\n", Math.exp( 2.0 ) );
      System.out.printf( "Math.floor( 9.2 ) = %f\n", Math.floor( 9.2 ) );
      System.out.printf( "Math.floor( -9.8 ) = %f\n",
         Math.floor( -9.8 ) );
      System.out.printf( "Math.log( Math.E ) = %f\n",
         Math.log( Math.E ) );
      System.out.printf( "Math.log( Math.E * Math.E ) = %f\n",
         Math.log( Math.E * Math.E ) );
```

```
      System.out.printf( "Math.max( 2.3, 12.7 ) = %f\n",
         Math.max( 2.3, 12.7 ) );
      System.out.printf( "Math.max( -2.3, -12.7 ) = %f\n",
         Math.max( -2.3, -12.7 ) );
      System.out.printf( "Math.min( 2.3, 12.7 ) = %f\n",
         Math.min( 2.3, 12.7 ) );
      System.out.printf( "Math.min( -2.3, -12.7 ) = %f\n",
         Math.min( -2.3, -12.7 ) );
      System.out.printf( "Math.pow( 2.0, 7.0 ) = %f\n",
         Math.pow( 2.0, 7.0 ) );
      System.out.printf( "Math.pow( 9.0, 0.5 ) = %f\n",
         Math.pow( 9.0, 0.5 ) );
      System.out.printf( "Math.sin( 0.0 ) = %f\n", Math.sin( 0.0 ) );
      System.out.printf( "Math.sqrt( 900.0 ) = %f\n",
         Math.sqrt( 900.0 ) );
      System.out.printf( "Math.sqrt( 9.0 ) = %f\n", Math.sqrt( 9.0 ) );
      System.out.printf( "Math.tan( 0.0 ) = %f\n", Math.tan( 0.0 ) );
   } // fin de main
} // fin de la clase PruebaMath
```

```
Math.abs( 23.7 ) = 23.700000
Math.abs( 0.0 ) = 0.000000
Math.abs( -23.7 ) = 23.700000
Math.ceil( 9.2 ) = 10.000000
Math.ceil( -9.8 ) = -9.000000
Math.cos( 0.0 ) = 1.000000
Math.exp( 1.0 ) = 2.718282
Math.exp( 2.0 ) = 7.389056
Math.floor( 9.2 ) = 9.000000
Math.floor( -9.8 ) = -10.000000
Math.log( Math.E ) = 1.000000
Math.log( Math.E * Math.E ) = 2.000000
Math.max( 2.3, 12.7 ) = 12.700000
Math.max( -2.3, -12.7 ) = -2.300000
Math.min( 2.3, 12.7 ) = 2.300000
Math.min( -2.3, -12.7 ) = -12.700000
Math.pow( 2.0, 7.0 ) = 128.000000
Math.pow( 9.0, 0.5 ) = 3.000000
Math.sin( 0.0 ) = 0.000000
Math.sqrt( 900.0 ) = 30.000000
Math.sqrt( 9.0 ) = 3.000000
Math.tan( 0.0 ) = 0.000000
```

6.4 a) `double hipotenusa( double lado1, double lado2 )`
b) `int menor( int x, int y, int z )`
c) `void instrucciones()`
d) `float intAFloat( int numero )`

6.5 a) Error: el método `h` está declarado dentro del método `g`.
Corrección: mueva la declaración de `h` fuera de la declaración de `g`.
b) Error: se supone que el método debe devolver un entero, pero no es así.
Corrección: elimine la variable `resultado`, y coloque la instrucción

```
return x + y;
```

en el método, o agregue la siguiente instrucción al final del cuerpo del método:

```
return resultado;
```

c) Error: el punto y coma que va después del paréntesis derecho de la lista de parámetros es incorrecto, y el parámetro `a` no debe volver a declararse en el método.
Corrección: elimine el punto y coma que va después del paréntesis derecho de la lista de parámetros, y elimine la declaración `float a;`.

d) Error: el método devuelve un valor cuando no debe hacerlo.
Corrección: cambie el tipo de valor de retorno de `void` a `int`.

6.6 La siguiente solución calcula el volumen de una esfera, utilizando el radio introducido por el usuario:

```
// Ejercicio 6.6: Esfera.java
// Calcula el volumen de una esfera.
import java.util.Scanner;

public class Esfera
{
   // obtiene el radio del usuario y muestra el volumen de la esfera
   public void determinarVolumenEsfera()
   {
      Scanner entrada = new Scanner( System.in );

      System.out.print( "Escriba el radio de la esfera: " );
      double radio = entrada.nextDouble();

      System.out.printf( "El volumen es %f\n", volumenEsfera( radio ) );
   } // fin del método determinarVolumenEsfera

   // calcula y devuelve el volumen de una esfera
   public double volumenEsfera( double radio )
   {
      double volumen = ( 4.0 / 3.0 ) * Math.PI * Math.pow( radio, 3 );
      return volumen;
   } // fin del método volumenEsfera
} // fin de la clase Esfera
```

```
// Ejercicio 6.6: PruebaEsfera.java
// Calcula el volumen de una esfera.

public class PruebaEsfera
{
   // punto de inicio de la aplicación
   public static void main( String args[] )
   {
      Esfera miEsfera = new Esfera();
      miEsfera.determinarVolumenEsfera();
   } // fin de main
} // fin de la clase PruebaEsfera
```

```
Escriba el radio de la esfera: 4
El volumen es 268.082573
```

Ejercicios

6.7 ¿Cuál es el valor de x después de que se ejecuta cada una de las siguientes instrucciones?

a) `x = Math.abs( 7.5 );`
b) `x = Math.floor( 7.5 );`
c) `x = Math.abs( 0.0 );`
d) `x = Math.ceil( 0.0 );`
e) `x = Math.abs( -6.4 );`
f) `x = Math.ceil( -6.4 );`
g) `x = Math.ceil( -Math.abs( -8 + Math.floor( -5.5 ) ) );`

6.8 Un estacionamiento cobra una cuota mínima de $2.00 por estacionarse hasta tres horas. El estacionamiento cobra $0.50 adicionales por cada hora o *fracción* que se pase de tres horas. El cargo máximo para cualquier periodo dado de 24 horas es de $10.00. Suponga que ningún automóvil se estaciona durante más de 24 horas a la vez. Escriba una aplicación que calcule y muestre los cargos por estacionamiento para cada cliente que se haya estacionado ayer. Debe introducir las horas de estacionamiento para cada cliente. El programa debe mostrar el cargo para el cliente actual y debe calcular y mostrar el total corriente de los recibos de ayer. El programa debe utilizar el método `calcularCargos` para determinar el cargo para cada cliente.

6.9 Una aplicación del método `Math.floor` es redondear un valor al siguiente entero. La instrucción

```
y = Math.floor( x + 0.5 );
```

redondea el número `x` al entero más cercano y asigna el resultado a `y`. Escriba una aplicación que lea valores `double` y que utilice la instrucción anterior para redondear cada uno de los números a su entero más cercano. Para cada número procesado, muestre tanto el número original como el redondeado.

6.10 `Math.floor` puede utilizarse para redondear un número hasta un lugar decimal específico. La instrucción

```
y = Math.floor( x * 10 + 0.5 ) / 10;
```

redondea `x` en la posición de las décimas (es decir, la primera posición a la derecha del punto decimal). La instrucción

```
y = Math.floor( x * 100 + 0.5 ) / 100;
```

redondea `x` en la posición de las centésimas (es decir, la segunda posición a la derecha del punto decimal). Escriba una aplicación que defina cuatro métodos para redondear un número `x` en varias formas:

a) `redondearAInteger( numero )`
b) `redondearADecimas( numero )`
c) `redondearACentesimas( numero )`
d) `redondearAMilesimas( numero )`

Para cada valor leído, su programa debe mostrar el valor original, el número redondeado al entero más cercano, el número redondeado a la décima más cercana, el número redondeado a la centésima más cercana y el número redondeado a la milésima más cercana.

6.11 Responda a cada una de las siguientes preguntas:
a) ¿Qué significa elegir números “al azar”?
b) ¿Por qué es el método `nextInt` de la clase `Random` útil para simular juegos al azar?
c) ¿Por qué es a menudo necesario escalar o desplazar los valores producidos por un objeto `Random`?
d) ¿Por qué es la simulación computarizada de las situaciones reales una técnica útil?

6.12 Escriba instrucciones que asignen enteros aleatorios a la variable n en los siguientes rangos:
a) $1 \le n \le 2$.
b) $1 \le n \le 100$.
c) $0 \le n \le 9$.
d) $1000 \le n \le 1112$.
e) $-1 \le n \le 1$.
f) $-3 \le n \le 11$.

6.13 Para cada uno de los siguientes conjuntos de enteros, escriba una sola instrucción que imprima un número al azar del conjunto:
a) 2, 4, 6, 8, 10.
b) 3, 5, 7, 9, 11.
c) 6, 10, 14, 18, 22.

6.14 Escriba un método llamado `enteroPotencia( base, exponente )` que devuelva el valor de

$$base^{exponente}$$

Por ejemplo, `enteroPotencia( 3, 4 )` calcula 3^4 (o `3 * 3 * 3 * 3`). Suponga que `exponente` es un entero positivo distinto de cero y que `base` es un entero. El método `enteroPotencia` debe utilizar un ciclo `for` o `while` para controlar el cálculo. No utilice ningún método de la biblioteca de matemáticas. Incorpore este método en una aplicación que lea valores enteros para `base` y `exponente`, y que realice el cálculo con el método `enteroPotencia`.

6.15 Defina un método llamado `hipotenusa` que calcule la longitud de la hipotenusa de un triángulo rectángulo, cuando se proporcionen las longitudes de los otros dos lados. (Utilice los datos de ejemplo de la figura 6.26.) El método debe tomar dos argumentos de tipo `double` y devolver la hipotenusa como un valor `double`. Incorpore este método en una aplicación que lea los valores para `lado1` y `lado2`, y que realice el cálculo con el método `hipotenusa`. Determine la longitud de la hipotenusa para cada uno de los triángulos de la figura 6.26.

6.16 Escriba un método llamado `multiplo` que determine, para un par de enteros, si el segundo entero es múltiplo del primero. El método debe tomar dos argumentos enteros y devolver `true` si el segundo es múltiplo del primero, y `false` en caso contrario. [*Sugerencia:* utilice el operador residuo]. Incorpore este método en una aplicación que reciba como entrada una serie de pares de enteros (un par a la vez) y determine si el segundo valor en cada par es un múltiplo del primero.

6.17 Escriba un método llamado `esPar` que utilice el operador residuo (`%`) para determinar si un entero dado es par. El método debe tomar un argumento entero y devolver `true` si el entero es par, y `false` en caso contrario. Incorpore este método en una aplicación que reciba como entrada una secuencia de enteros (uno a la vez), y que determine si cada uno es par o impar.

6.18 Escriba un método llamado `cuadradoDeAsteriscos` que muestre un cuadrado relleno (el mismo número de filas y columnas) de asteriscos cuyo lado se especifique en el parámetro entero `lado`. Por ejemplo, si `lado` es `4`, el método debe mostrar:

```
****
****
****
****
```

Incorpore este método a una aplicación que lea un valor entero para el parámetro `lado` que teclea el usuario, y despliegue los asteriscos con el método `cuadradoDeAsteriscos`.

6.19 Modifique el método creado en el ejercicio 6.18 para formar el cuadrado de cualquier carácter que esté contenido en el parámetro tipo carácter `caracterRelleno`. Por ejemplo, si `lado` es `5` y `caracterRelleno` es “`#`”, el método debe imprimir

```
####
####
####
####
####
```

6.20 Escriba una aplicación que pida al usuario el radio de un círculo y que utilice un método llamado `circuloArea` para calcular e imprimir el área de ese círculo.

6.21 Escriba segmentos de programas que realicen cada una de las siguientes tareas:

a) Calcular la parte entera del cociente, cuando el entero `a` se divide entre el entero `b`.
b) Calcular el residuo entero cuando el entero `a` se divide entre el entero `b`.
c) Utilizar las piezas de los programas desarrollados en las partes (a) y (b) para escribir un método llamado `mostrarDigitos`, que reciba un entero entre `1` y `99999`, y que lo muestre como una secuencia de dígitos, separando cada par de dígitos por dos espacios. Por ejemplo, el entero `4562` debe aparecer como

```
4  5  6  2
```

d) Incorpore el método desarrollado en la parte (c) en una aplicación que reciba como entrada un entero y que llame al método `mostrarDigitos`, pasándole a este método el entero introducido. Muestre los resultados.

Triángulo	Lado 1	Lado 2
1	3.0	4.0
2	5.0	12.0
3	8.0	15.0

figura 6.26 | Valores para los lados de los triángulos del ejercicio 6.15.

6.22 Implemente los siguientes métodos enteros:

a) El método `centigrados` que devuelve la equivalencia en grados centígrados de una temperatura en grados fahrenheit, utilizando el cálculo

```
centigrados = 5.0 / 9.0 * ( fahrenheit - 32 );
```

b) El método `fahrenheit` que devuelve la equivalencia en grados fahrenheit de una temperatura en grados centígrados, utilizando el cálculo

```
fahrenheit = 9.0 / 5.0 * centigrados + 32;
```

c) Utilice los métodos de las partes (a) y (b) para escribir una aplicación que permita al usuario, ya sea escribir una temperatura en grados fahrenheit y mostrar su equivalente en grados centígrados, o escribir una temperatura en grados centígrados y mostrar su equivalente en grados fahrenheit.

6.23 Escriba un método llamado `minimo3` que devuelva el menor de tres números de punto flotante. Use el método `Math.min` para implementar `minimo3`. Incorpore el método en una aplicación que reciba como entrada tres valores por parte del usuario, determine el valor menor y muestre el resultado.

6.24 Se dice que un número entero es un *número perfecto* si sus factores, incluyendo 1 (pero no el número entero), al sumarse dan como resultado el número entero. Por ejemplo, 6 es un número perfecto ya que 6 = 1 + 2 + 3. Escriba un método llamado `perfecto` que determine si el parámetro `numero` es un número perfecto. Use este método en una aplicación que determine y muestre todos los números perfectos entre 1 y 1000. Imprima los factores de cada número perfecto para confirmar que el número sea realmente perfecto. Ponga a prueba el poder de su computadora, evaluando números más grandes que 1000. Muestre los resultados.

6.25 Se dice que un entero es *primo* si puede dividirse solamente por 1 y por sí mismo. Por ejemplo, 2, 3, 5 y 7 son primos, pero 4, 6, 8 y 9 no.

a) Escriba un método que determine si un número es primo.

b) Use este método en una aplicación que determine e imprima todos los números primos menores que 10,000. ¿Cuántos números hasta 10,000 tiene que probar para asegurarse de encontrar todos los números primos?

c) Al principio podría pensarse que *n*/2 es el límite superior para evaluar si un número es primo, pero lo máximo que se necesita es ir hasta la raíz cuadrada de *n*. ¿Por qué? Vuelva a escribir el programa y ejecútelo de ambas formas.

6.26 Escriba un método que tome un valor entero y devuelva el número con sus dígitos invertidos. Por ejemplo, para el número 7631, el método debe regresar 1367. Incorpore el método en una aplicación que reciba como entrada un valor del usuario y muestre el resultado.

6.27 El *máximo común divisor (MCD)* de dos enteros es el entero más grande que puede dividir uniformemente a cada uno de los dos números. Escriba un método llamado `mcd` que devuelva el máximo común divisor de dos enteros. [*Sugerencia:* tal vez sea conveniente que utilice el algoritmo de Euclides. Puede encontrar información acerca de este algoritmo en `es.wikipedia.org/wiki/Algoritmo_de_Euclides`]. Incorpore el método en una aplicación que reciba como entrada dos valores del usuario y muestre el resultado.

6.28 Escriba un método llamado `puntosCalidad` que reciba como entrada el promedio de un estudiante y devuelva `4` si el promedio se encuentra entre 90 y 100, `3` si el promedio se encuentra entre 80 y 89, `2` si el promedio se encuentra entre 70 y 79, `1` si el promedio se encuentra entre 60 y 69, y 0 si el promedio es menor de 60. Incorpore el método en una aplicación que reciba como entrada un valor del usuario y muestre el resultado.

6.29 Escriba una aplicación que simule el lanzamiento de monedas. Deje que el programa lance una moneda cada vez que el usuario seleccione la opción del menú "`Lanzar moneda`". Cuente el número de veces que aparezca cada uno de los lados de la moneda. Muestre los resultados. El programa debe llamar a un método separado, llamado `tirar`, que no tome argumentos y devuelva `false` en caso de cara, y `true` en caso de cruz. [*Nota:* si el programa simula en forma realista el lanzamiento de monedas, cada lado de la moneda debe aparecer aproximadamente la mitad del tiempo.]

6.30 Las computadoras están tomando un papel cada vez más importante en la educación. Escriba un programa que ayude a un estudiante de escuela primaria, para que aprenda a multiplicar. Use un objeto `Random` para producir dos enteros positivos de un dígito. El programa debe entonces mostrar una pregunta al usuario, como:

```
¿Cuánto es 6 por 7?
```

El estudiante entonces debe escribir la respuesta. Luego, el programa debe verificar la respuesta del estudiante. Si es correcta, dibuje la cadena `"Muy bien!"` y haga otra pregunta de multiplicación. Si la respuesta es incorrecta, dibuje la cadena `"No. Por favor intenta de nuevo."` y deje que el estudiante intente la misma pregunta varias veces, hasta que esté correcta. Debe utilizarse un método separado para generar cada pregunta nueva. Este método debe llamarse una vez cuando la aplicación empiece a ejecutarse, y cada vez que el usuario responda correctamente a la pregunta.

6.31 El uso de las computadoras en la educación se conoce como *instrucción asistida por computadora* (*CAI,* por sus siglas en inglés). Un problema que se desarrolla en los entornos CAI es la fatiga de los estudiantes. Este problema puede eliminarse si se varía el diálogo de la computadora para mantener la atención del estudiante. Modifique el programa del ejercicio 6.30 de manera que los diversos comentarios se impriman para cada respuesta correcta e incorrecta, de la siguiente manera:

Contestaciones a una respuesta correcta:

```
Muy bien!
Excelente!
Buen trabajo!
Sigue asi!
```

Contestaciones a una respuesta incorrecta:

```
No. Por favor intenta de nuevo.
Incorrecto. Intenta una vez mas.
No te rindas!
No. Sigue intentando.
```

Use la generación de números aleatorios para elegir un número entre 1 y 4 que se utilice para seleccionar una contestación apropiada a cada respuesta. Use una instrucción `switch` para emitir las contestaciones.

6.32 Los sistemas de instrucción asistida por computadora más sofisticados supervisan el rendimiento del estudiante durante cierto tiempo. La decisión de empezar un nuevo tema se basa a menudo en el éxito del estudiante con los temas anteriores. Modifique el programa del ejercicio 6.31 para contar el número de respuestas correctas e incorrectas por parte del estudiante. Una vez que el estudiante escriba 10 respuestas, su programa debe calcular el porcentaje de respuestas correctas. Si éste es menor del 75%, imprima `Por favor pida ayuda adicional a su instructor` y reinicie el programa, para que otro estudiante pueda probarlo.

6.33 Escriba una aplicación que juegue a "adivina el número" de la siguiente manera: su programa elige el número a adivinar, seleccionando un entero aleatorio en el rango de 1 a 1000. La aplicación muestra el indicador `Adivine un número entre 1 y 1000`. El jugador escribe su primer intento. Si la respuesta del jugador es incorrecta, su programa debe mostrar el mensaje `Demasiado alto. Intente de nuevo.` o `Demasiado bajo. Intente de nuevo.`, para ayudar a que el jugador "se acerque" a la respuesta correcta. El programa debe pedir al usuario que escriba su siguiente intento. Cuando el usuario escriba la respuesta correcta, muestre el mensaje `Felicidades. Adivino el numero!` y permita que el usuario elija si desea jugar otra vez. [*Nota:* la técnica para adivinar empleada en este problema es similar a una búsqueda binaria, que veremos en el capítulo 16, Búsqueda y ordenamiento].

6.34 Modifique el programa del ejercicio 6.33 para contar el número de intentos que haga el jugador. Si el número es 10 o menos, imprima el mensaje `O ya sabia usted el secreto, o tuvo suerte!` Si el jugador adivina el número en 10 intentos, imprima el mensaje `Aja! Sabia usted el secreto!` Si el jugador hace más de 10 intentos, imprima el mensaje `Deberia haberlo hecho mejor!` ¿Por qué no se deben requerir más de 10 intentos? Bueno, en cada "buen intento", el jugador debe poder eliminar la mitad de los números, después la mitad de los números restantes, y así en lo sucesivo.

6.35 En los ejercicios 6.30 al 6.32 se desarrolló un programa de instrucción asistida por computadora para enseñar a un estudiante de escuela primara cómo multiplicar. Realice las siguientes mejoras:

a) Modifique el programa para que permita al usuario introducir un nivel de capacidad escolar. Un nivel de 1 significa que el programa debe usar sólo números de un dígito en los problemas, un nivel 2 significa que el programa debe utilizar números de dos dígitos máximo, etcétera.

b) Modifique el programa para permitir al usuario que elija el tipo de problemas aritméticos que desea estudiar. Una opción de `1` significa problemas de suma solamente, `2` significa problemas de resta, `3` significa problemas de multiplicación, `4` significa problemas de división y `5` significa una mezcla aleatoria de problemas de todos estos tipos.

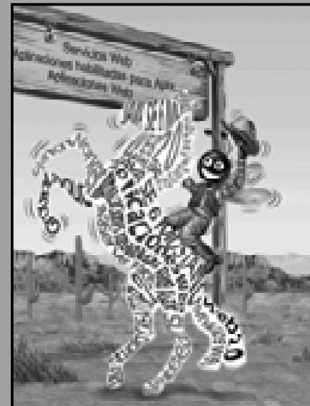
6.36 Escriba un método llamado `distancia`, para calcular la distancia entre dos puntos (*x1*, *y1*) y (*x2*, *y2*). Todos los números y valores de retorno deben ser de tipo `double`. Incorpore este método en una aplicación que permita al usuario introducir las coordenadas de los puntos.

6.37 Modifique el programa Craps de la figura 6.9 para permitir apuestas. Inicialice la variable `saldoBanco` con $1000. Pida al jugador que introduzca una `apuesta`. Compruebe que esa `apuesta` sea menor o igual al `saldoBanco` y, si no lo es, haga que el usuario vuelva a introducir la `apuesta` hasta que se introduzca un valor válido. Después de esto, comience un juego de craps. Si el jugador gana, agregue la `apuesta` al `saldoBanco` e imprima el nuevo `saldoBanco`. Si el jugador pierde, reste la `apuesta` al `saldoBanco`, imprima el nuevo `saldoBanco`, compruebe si `saldoBanco` se ha vuelto cero y, de ser así, imprima el mensaje `"Lo siento. Se quedo sin fondos!"` A medida que el juego progrese, imprima varios mensajes para crear algo de "charla", como `"Oh, se esta yendo a la quiebra, verdad?"`, o `"Oh, vamos, arriesguese!"`, o `"La hizo en grande. Ahora es tiempo de cambiar sus fichas por efectivo!"`. Implemente la "charla" como un método separado que seleccione en forma aleatoria la cadena a mostrar.

6.38 Escriba una aplicación que muestre una tabla de los equivalentes en binario, octal y hexadecimal de los números decimales en el rango de 1 al 256. Si no está familiarizado con estos sistemas numéricos, lea el apéndice E primero.

7

Arreglos



Ahora ve, escríbelo ante ellos en una tabla, y anótalo en un libro.

—Isaías 30:8

Ir más allá es tan malo como no llegar.

—Confucio

Comienza en el principio… y continúa hasta que llegues al final; después detente.

—Lewis Carroll

OBJETIVOS

En este capítulo aprenderá a:

- Conocer qué son los arreglos.
- Utilizar arreglos para almacenar datos en, y obtenerlos de listas y tablas de valores.
- Declarar arreglos, inicializarlos y hacer referencia a elementos individuales de los arreglos.
- Utilizar la instrucción `for` mejorada para iterar a través de los arreglos.
- Pasar arreglos a los métodos.
- Declarar y manipular arreglos multidimensionales.
- Escribir métodos que utilicen listas de argumentos de longitud variable.
- Leer los argumentos de línea de comandos en un programa.

Plan general

7.1 Introducción

En este capítulo presentamos el importante tema de las **estructuras de datos**: colecciones de elementos de datos relacionados. Los **arreglos** son estructuras de datos que consisten de elementos de datos relacionados, del mismo tipo. Los arreglos son entidades de longitud fija; conservan la misma longitud una vez creados, aunque puede reasignarse una variable tipo arreglo de tal forma que haga referencia a un nuevo arreglo de distinta longitud. En los capítulos 17 a 19 estudiaremos con detalle las estructuras de datos.

Después de hablar acerca de cómo se declaran, crean y inicializan los arreglos, presentaremos una serie de ejemplos prácticos que demuestran varias manipulaciones comunes de los arreglos. También presentaremos un ejemplo práctico en el que se examina la forma en que los arreglos pueden ayudar a simular los procesos de barajar y repartir cartas, para utilizarlos en una aplicación que implementa un juego de cartas. Después presentaremos la instrucción `for` mejorada de java, la cual permite que un programa acceda a los datos en un arreglo con más facilidad que la instrucción `for` controlada por contador, que presentamos en la sección 5.3. Hay dos secciones de este capítulo en las que se amplía el ejemplo práctico de la clase `LibroCalificaciones` de los capítulos 3 a 5. En especial, utilizaremos los arreglos para permitir que la clase mantenga un conjunto de calificaciones en memoria y analizar las calificaciones que obtuvieron los estudiantes en distintos exámenes en un semestre; dos herramientas que no están presentes en las versiones anteriores de la clase. Éstos y otros ejemplos del capítulo demostrarán las formas en las que los arreglos permiten a los programadores organizar y manipular datos.

7.2 Arreglos

En Java, un arreglo es un grupo de variables (llamadas **elementos** o **componentes**) que contienen valores, todos del mismo tipo. Recuerde que los tipos en Java se dividen en dos categorías: tipos primitivos y tipos de referencia. Los arreglos son objetos, por lo que se consideran como tipos de referencia. Como veremos pronto, lo que consideramos generalmente como un arreglo es en realidad una referencia a un objeto arreglo en memoria. Los elementos de un arreglo pueden ser tipos primitivos o de referencia (incluyendo arreglos, como veremos en la sección 7.9). Para hacer referencia a un elemento específico en un arreglo, debemos especificar el nombre de la referencia al arreglo y el número de la posición del elemento en el arreglo. El número de la posición del elemento se conoce formalmente como el **índice** o **subíndice** del elemento.

En la figura 7.1 se muestra una representación lógica de un arreglo de enteros, llamado `c`. Este arreglo contiene 12 elementos. Un programa puede hacer referencia a cualquiera de estos elementos mediante una **expresión de acceso a un arreglo** que incluye el nombre del arreglo, seguido por el índice del elemento específico encerrado

entre **corchetes** ([]). El primer elemento en cualquier arreglo tiene el **índice cero**, y algunas veces se le denomina **elemento cero**. Por lo tanto, los elementos del arreglo c son c[0], c[1], c[2], y así en lo sucesivo. El mayor índice en el arreglo c es 11: 1 menos que 12, el número de elementos en el arreglo. Los nombres de los arreglos siguen las mismas convenciones que los demás nombres de variables.

Un índice debe ser un entero positivo. Un programa puede utilizar una expresión como índice. Por ejemplo, si suponemos que la variable a es 5 y que b es 6, entonces la instrucción

```
c[ a + b ] += 2;
```

suma 2 al elemento c[11] del arreglo. Observe que el nombre del arreglo con subíndice es una expresión de acceso al arreglo. Dichas expresiones pueden utilizarse en el lado izquierdo de una asignación, para colocar un nuevo valor en un elemento del arreglo.

Nombre del arreglo (c)

c[0]	-45
c[1]	6
c[2]	0
c[3]	72
c[4]	1543
c[5]	-89
c[6]	0
c[7]	62
c[8]	-3
c[9]	1
c[10]	6453
c[11]	78

Índice (o subíndice) del elemento en el arreglo c

Figura 7.1 | Un arreglo con 12 elementos.



Error común de programación 7.1

Usar un valor de tipo long *como índice de un arreglo produce un error de compilación. Un índice debe ser un valor* int*, o un valor de un tipo que pueda promoverse a* int*; a saber,* byte*,* short *o* char*, pero no* long*.*

Examinaremos el arreglo c de la figura 7.1 con más detalle. El **nombre** del arreglo es c. Cada instancia de un objeto conoce su propia longitud y mantiene esta información en un campo length. La expresión c.length accede al campo length del arreglo c para determinar la longitud del arreglo. Observe que, aun cuando el miembro length de un arreglo es public, no puede cambiarse, ya que es una variable final. La manera en que se hace referencia a los 12 elementos de este arreglo es: c[0], c[1], c[2], …, c[11]. El valor de c[0] es -45, el valor de c[1] es 6, el de c[2] es 0, el de c[7] es 62 y el de c[11] es 78. Para calcular la suma de los valores contenidos en los primeros tres elementos del arreglo c y almacenar el resultado en la variable suma, escribiríamos lo siguiente:

```
suma = c[ 0 ] + c[ 1 ] + c[ 2 ];
```

Para dividir el valor de c[6] entre 2 y asignar el resultado a la variable x, escribiríamos lo siguiente:

```
x = c[ 6 ] / 2;
```

7.3 Declaración y creación de arreglos

Los objetos arreglo ocupan espacio en memoria. Al igual que los demás objetos, los arreglos se crean con la palabra clave new. Para crear un objeto arreglo, el programador especifica el tipo de cada elemento y el número

de elementos que se requieren para el arreglo, como parte de una **expresión para crear un arreglo** que utiliza la palabra clave `new`. Dicha expresión devuelve una referencia que puede almacenarse en una variable tipo arreglo. La siguiente declaración y expresión crea un objeto arreglo, que contiene 12 elementos `int`, y almacena la referencia del arreglo en la variable `c`:

```
int c[] = new int[ 12 ];
```

Esta expresión puede usarse para crear el arreglo que se muestra en la figura 7.1. Esta tarea también puede realizarse en dos pasos, como se muestra a continuación:

```
int c[ ];           // declara la variable arreglo
c = new int[ 12 ];  // crea el arreglo; lo asigna a la variable tipo arreglo
```

En la declaración, los corchetes que van después del nombre de la variable `c` indican que `c` es una variable que hará referencia a un arreglo de valores `int` (es decir, `c` almacenará una referencia a un objeto arreglo). En la instrucción de asignación, la variable arreglo `c` recibe la referencia a un nuevo objeto arreglo de 12 elementos `int`. Al crear un arreglo, cada uno de sus elementos recibe un valor predeterminado: cero para los elementos numéricos de tipos primitivos, `false` para los elementos `boolean` y `null` para las referencias (cualquier tipo no primitivo). Como pronto veremos, podemos proporcionar valores iniciales para los elementos no específicos ni predeterminados al crear un arreglo.



Error común de programación 7.2

En la declaración de un arreglo, si se especifica el número de elementos en los corchetes de la declaración (por ejemplo, `int c[ 12 ];`*) se produce un error de sintaxis.*

Un programa puede crear varios arreglos en una sola declaración. La siguiente declaración de un arreglo `String` reserva 100 elementos para `b` y 27 para `x`:

```
String b[] = new String[ 100 ], x[] = new String[ 27 ];
```

En este caso, se aplica el nombre de la clase `String` a cada variable en la declaración. Por cuestión de legibilidad, es preferible declarar sólo una variable en cada declaración, como en:

```
String b[] = new string[ 100 ]; // crea el arreglo b
String x[] = new string[ 27 ]; // crea el arreglo x
```



Buena práctica de programación 7.1

Por cuestión de legibilidad, declare sólo una variable en cada declaración. Mantenga cada declaración en una línea separada e incluya un comentario que describa a la variable que está declarando.

Cuando se declara un arreglo, su tipo y los corchetes pueden combinarse al principio de la declaración para indicar que todos los identificadores en la declaración son variables tipo arreglo. Por ejemplo, la declaración

```
double[] arreglo1, arreglo2;
```

indica que `arreglo1` y `arreglo2` son variables tipo "arreglo de `double`". La anterior declaración es equivalente a:

```
double arreglo1[];
double arreglo2[];
```

o

```
double[] arreglo1;
double[] arreglo2;
```

Los pares anteriores de declaraciones son equivalentes; cuando se declara sólo una variable en cada declaración, los corchetes pueden colocarse ya sea antes del tipo, o después del nombre de la variable tipo arreglo.



Error común de programación 7.3

Declarar múltiples variables tipo arreglo en una sola declaración puede provocar errores sutiles. Considere la declaración `int[] a, b, c;`. Si `a`, `b` y `c` deben declararse como variables tipo arreglo, entonces esta declaración es correcta; al colocar corchetes directamente después del tipo, indicamos que todos los identificadores en la declaración son variables tipo arreglo. No obstante, si sólo `a` debe ser una variable tipo arreglo, y `b` y `c` deben ser variables `int` individuales, entonces esta declaración es incorrecta; la declaración `int a[], b, c;` lograría el resultado deseado.

Un programa puede declarar arreglos de cualquier tipo. Cada elemento de un arreglo de tipo primitivo contiene un valor del tipo declarado del arreglo. De manera similar, en un arreglo de un tipo de referencia, cada elemento es una referencia a un objeto del tipo declarado del arreglo. Por ejemplo, cada elemento de un arreglo `int` es un valor `int`, y cada elemento de un arreglo `String` es una referencia a un objeto `String`.

7.4 Ejemplos acerca del uso de los arreglos

En esta sección presentaremos varios ejemplos que muestran cómo declarar, crear e inicializar arreglos y cómo manipular sus elementos.

Cómo crear e inicializar un arreglo

En la aplicación de la figura 7.2 se utiliza la palabra clave `new` para crear un arreglo de 10 elementos `int`, los cuales inicialmente tienen el valor cero (el valor predeterminado para las variables `int`).

En la línea 8 se declara `arreglo`, una referencia capaz de referirse a un arreglo de elementos `int`. En la línea 10 se crea el objeto arreglo y se asigna su referencia a la variable `arreglo`. La línea 12 imprime los encabezados de las columnas. La primera columna representa el índice (0 a 9) para cada elemento del arreglo, y la segunda el valor predeterminado (0) de cada elemento del arreglo.

```
1  // Fig. 7.2: InicArreglo.java
2  // Creación de un arreglo.
3
4  public class InicArreglo
5  {
6     public static void main( String args[] )
7     {
8        int arreglo[]; // declara un arreglo con el mismo nombre
9
10       arreglo = new int[ 10 ]; // crea el espacio para el arreglo
11
12       System.out.printf( "%s%8s\n", "Indice", "Valor" ); // encabezados de columnas
13
14       // imprime el valor de cada elemento del arreglo
15       for ( int contador = 0; contador < arreglo.length; contador++ )
16          System.out.printf( "%5d%8d\n", contador, arreglo[ contador ] );
17    } // fin de main
18 } // fin de la clase InicArreglo
```

```
Indice   Valor
    0       0
    1       0
    2       0
    3       0
    4       0
    5       0
    6       0
    7       0
    8       0
    9       0
```

Figura 7.2 | Inicialización de los elementos de un arreglo con valores predeterminados de cero.

La instrucción `for` en las líneas 15 y 16 imprime el número de índice (representado por `contador`) y el valor de cada elemento del arreglo (representado por `arreglo[ contador ]`). Observe que al principio la variable de control del ciclo `contador` es `0` (los valores de los índices empiezan en 0, por lo que al utilizar un conteo con base cero se permite al ciclo acceder a todos los elementos del arreglo. La condición de continuación de ciclo de la instrucción `for` utiliza la expresión `arreglo.length` (línea 15) para determinar la longitud del arreglo. En este ejemplo la longitud del arreglo es de 10, por lo que el ciclo continúa ejecutándose mientras el valor de la variable de control `contador` sea menor que 10. El valor más alto para el subíndice de un arreglo de 10 elementos es 9, por lo que al utilizar el operador "menor que" en la condición de continuación de ciclo se garantiza que el ciclo no trate de acceder a un elemento más allá del final del arreglo (es decir, durante la iteración final del ciclo, `contador` es `9`). Pronto veremos lo que hace Java cuando encuentra un subíndice fuera de rango en tiempo de ejecución.

Uso de un inicializador de arreglo

Un programa puede crear un arreglo e inicializar sus elementos con un **inicializador de arreglo**, que es una lista de expresiones separadas por comas (la cual se conoce también como **lista inicializadora**) encerrada entre llaves (`{` y `}`); la longitud del arreglo se determina en base al número de elementos en la lista inicializadora. Por ejemplo, la declaración

```
int n[] = { 10, 20, 30, 40, 50 };
```

crea un arreglo de cinco elementos con los valores de índices `0`, `1`, `2`, `3` y `4`. El elemento `n[ 0 ]` se inicializa con `10`, `n[ 1 ]` se inicializa con `20`, y así en lo sucesivo. Esta declaración no requiere que `new` cree el objeto arreglo. Cuando el compilador encuentra la declaración de un arreglo que incluye una lista inicializadora, cuenta el número de inicializadores en la lista para determinar el tamaño del arreglo, y después establece la operación `new` apropiada "detrás de las cámaras".

La aplicación de la figura 7.3 inicializa un arreglo de enteros con 10 valores (línea 9) y muestra el arreglo en formato tabular. El código para mostrar los elementos del arreglo (líneas 14 y 15) es idéntico al de la figura 7.2 (líneas 15 y 16).

```
1  // Fig. 7.3: InicArreglo.java
2  // Inicialización de los elementos de un arreglo con un inicializador de arreglo.
3
4  public class InicArreglo
5  {
6     public static void main( String args[] )
7     {
8        // la lista inicializadora especifica el valor para cada elemento
9        int arreglo[] = { 32, 27, 64, 18, 95, 14, 90, 70, 60, 37 };
10
11       System.out.printf( "%s%8s\n", "Indice", "Valor" ); // encabezados de columnas
12
13       // imprime el valor del elemento de cada arreglo
14       for ( int contador = 0; contador < arreglo.length; contador++ )
15          System.out.printf( "%5d%8d\n", contador, arreglo[ contador ] );
16    } // fin de main
17 } // fin de la clase InicArreglo
```

```
Indice   Valor
     0      32
     1      27
     2      64
     3      18
     4      95
     5      14
     6      90
```

Figura 7.3 | Inicialización de los elementos de un arreglo con un inicializador de arreglo. (Parte 1 de 2).

```
      7       70
      8       60
      9       37
```

Figura 7.3 | Inicialización de los elementos de un arreglo con un inicializador de arreglo. (Parte 2 de 2).

Cálculo de los valores a guardar en un arreglo

La aplicación de la figura 7.4 crea un arreglo de 10 elementos y asigna a cada elemento uno de los enteros pares del 2 al 20 (2, 4, 6, …, 20). Después, la aplicación muestra el arreglo en formato tabular. La instrucción `for` en las líneas 12 y 13 calcula el valor de un elemento del arreglo, multiplicando el valor actual de la variable de control `contador` por 2, y después le suma 2.

La línea 8 utiliza el modificador `final` para declarar la **variable constante** `LONGITUD_ARREGLO` con el valor 10. Las variables constantes (también conocidas como variables `final`) deben inicializarse antes de usarlas, y no pueden modificarse de ahí en adelante. Si trata de modificar una variable `final` después de inicializarla en su declaración (como en la línea 8), el compilador genera el siguiente mensaje de error:

`cannot assign a value to final variable` *nombreVariable*

Si tratamos de acceder al valor de una variable `final` antes de inicializarla, el compilador produce el siguiente mensaje de error

`variable` *nombreVariable* `might not have been initialized`

```
1  // Fig. 7.4: InicArreglo.java
2  // Cálculo de los valores a colocar en los elementos de un arreglo.
3
4  public class InicArreglo
5  {
6     public static void main( String args[] )
7     {
8        final int LONGITUD_ARREGLO = 10; // declara la constante
9        int arreglo[] = new int[ LONGITUD_ARREGLO ]; // crea el arreglo
10
11       // calcula el valor para cada elemento del arreglo
12       for ( int contador = 0; contador < arreglo.length; contador++ )
13          arreglo[ contador ] = 2 + 2 * contador;
14
15       System.out.printf( "%s%8s\n", "Indice", "Valor" ); // encabezados de columnas
16
17       // imprime el valor de cada elemento del arreglo
18       for ( int contador = 0; contador < arreglo.length; contador++ )
19          System.out.printf( "%5d%8d\n", contador, arreglo[ contador ] );
20    } // fin de main
21 } // fin de la clase InicArreglo
```

```
Indice   Valor
     0       2
     1       4
     2       6
     3       8
     4      10
     5      12
     6      14
     7      16
     8      18
     9      20
```

Figura 7.4 | Cálculo de los valores a colocar en los elementos de un arreglo.



Buena práctica de programación 7.2

Las variables constantes también se conocen como ***constantes con nombre*** *o* ***variables de sólo lectura****. Con frecuencia, dichas variables mejoran la legibilidad de un programa, en comparación con los programas que utilizan valores literales (por ejemplo,* `10`*); una constante con nombre como* `LONGITUD_ARREGLO` *indica sin duda su propósito, mientras que un valor literal podría tener distintos significados, con base en el contexto en el que se utiliza.*



Error común de programación 7.4

Asignar un valor a una constante después de inicializarla es un error de compilación.



Error común de programación 7.5

Tratar de usar una constante antes de inicializarla es un error de compilación.

Suma de los elementos de un arreglo

A menudo, los elementos de un arreglo representan una serie de valores que se emplearán en un cálculo. Por ejemplo, si los elementos del arreglo representan las calificaciones de un examen, tal vez el profesor desee sumar el total de los elementos del arreglo y utilizar esa suma para calcular el promedio de la clase para el examen. Los ejemplos que utilizan la clase `LibroCalificaciones` más adelante en este capítulo, figura 7.14 y 7.18, utilizan esta técnica.

La aplicación de la figura 7.5 suma los valores contenidos en el arreglo entero de 10 elementos. El programa declara, crea e inicializa el arreglo en la línea 8. La instrucción `for` realiza los cálculos. [*Nota:* los valores suministrados como inicializadores de arreglos generalmente se introducen en un programa, en vez de especificarse en una lista inicializadora. Por ejemplo, una aplicación podría recibir los valores del usuario o de un archivo en disco (como veremos en el capítulo 14, Archivos y flujos). Al hacer que los datos se introduzcan como entrada en el programa éste se hace más flexible, ya que puede utilizarse con distintos conjuntos de datos].

Uso de gráficos de barra para mostrar los datos de un arreglo en forma gráfica

Muchas aplicaciones presentan datos a los usuarios en forma gráfica. Por ejemplo, con frecuencia los valores numéricos se muestran como barras en un gráfico de barras. En dicho gráfico, las barras más largas representan valores numéricos más grandes en forma proporcional. Una manera sencilla de mostrar los datos numéricos en forma gráfica es mediante un gráfico de barras que muestre cada valor numérico como una barra de asteriscos (*).

```
1  // Fig. 7.5: SumaArreglo.java
2  // Cálculo de la suma de los elementos de un arreglo.
3
4  public class SumaArreglo
5  {
6     public static void main( String args[] )
7     {
8        int arreglo[] = { 87, 68, 94, 100, 83, 78, 85, 91, 76, 87 };
9        int total = 0;
10
11       // suma el valor de cada elemento al total
12       for ( int contador = 0; contador < arreglo.length; contador++ )
13          total += arreglo[ contador ];
14
15       System.out.printf( "Total de los elementos del arreglo: %d\n", total );
16    } // fin de main
17 } // fin de la clase SumaArreglo
```

```
Total de los elementos del arreglo: 849
```

Figura 7.5 | Cálculo de la suma de los elementos de un arreglo.

A los profesores les gusta examinar a menudo la distribución de las calificaciones en un examen. Un profesor podría graficar el número de calificaciones en cada una de varias categorías, para visualizar la distribución de las calificaciones. Suponga que las calificaciones en un examen fueron 87, 68, 94, 100, 83, 78, 85, 91, 76 y 87. Observe que hubo una calificación de 100, dos calificaciones en el rango de 90 a 99, cuatro calificaciones en el rango de 80 a 89, dos en el rango de 70 a 79, una en el rango de 60 a 69 y ninguna por debajo de 60. Nuestra siguiente aplicación (figura 7.6) almacena estos datos de distribución de las calificaciones en un arreglo de 11 elementos, cada uno de los cuales corresponde a una categoría de calificaciones. Por ejemplo, `arreglo[ 0 ]` indica el número de calificaciones en el rango de 0 a 9, `arreglo[ 7 ]` indica el número de calificaciones en el rango de 70 a 79 y `arreglo[ 10 ]` indica el número de calificaciones de 100. Las dos versiones de la clase `LibroCalificaciones` que veremos más adelante en este capítulo (figuras 7.14 y 7.18) contienen código para calcular estas frecuencias de calificaciones, con base en un conjunto de calificaciones. Por ahora crearemos el arreglo en forma manual, mediante un análisis del conjunto de calificaciones.

```
// Fig. 7.6: GraficoBarras.java
// Programa para imprimir gráficos de barras.

public class GraficoBarras
{
   public static void main( String args[] )
   {
      int arreglo[] = { 0, 0, 0, 0, 0, 0, 1, 2, 4, 2, 1 };

      System.out.println( "Distribucion de calificaciones:" );

      // para cada elemento del arreglo, imprime una barra del gráfico
      for ( int contador = 0; contador < arreglo.length; contador++ )
      {
         // imprime etiqueta de la barra ( "00-09: ", ..., "90-99: ", "100: " )
         if ( contador == 10 )
            System.out.printf( "%5d: ", 100 );
         else
            System.out.printf( "%02d-%02d: ",
               contador * 10, contador * 10 + 9  );

         // imprime barra de asteriscos
         for ( int estrellas = 0; estrellas < arreglo[ contador ]; estrellas++ )
            System.out.print( "*" );

         System.out.println(); // inicia una nueva línea de salida
      } // fin de for externo
   } // fin de main
} // fin de la clase GraficoBarras
```

```
Distribucion de calificaciones:
00-09:
10-19:
20-29:
30-39:
40-49:
50-59:
60-69: *
70-79: **
80-89: ****
90-99: **
  100: *
```

Figura 7.6 | Programa para imprimir gráficos de barras.

La aplicación lee los números del arreglo y grafica la información en forma de un gráfico de barras. El programa muestra cada rango de calificaciones seguido de una barra de asteriscos, que indican el número de calificaciones en ese rango. Para etiquetar cada barra, las líneas 16 a 20 imprimen un rango de calificaciones (por ejemplo, `"70-79: "`) con base en el valor actual de `contador`. Cuando `contador` es `10`, la línea 17 imprime `100` con una anchura de campo de 5, seguida de dos puntos y un espacio, para alinear la etiqueta `"100: "` con las otras etiquetas de las barras. La instrucción `for` anidada (líneas 23 y 24) imprime las barras en pantalla. Observe la condición de continuación de ciclo en la línea 23 (`estrellas < arreglo[ contador ]`). Cada vez que el programa llega al `for` interno, el ciclo cuenta desde `0` hasta `arreglo[ contador ]`, con lo cual utiliza un valor en `arreglo` para determinar el número de asteriscos a mostrar en pantalla. En este ejemplo, los valores de `arreglo[ 0 ]` hasta `arreglo[ 5 ]` son `0`, ya que ningún estudiante recibió una calificación menor de 60. Por ende, el programa no muestra asteriscos enseguida de los primeros seis rangos de calificaciones. Observe que la línea 19 utiliza el especificador de formato `%02d` para imprimir los números en un rango de calificaciones. Este especificador indica que se debe dar formato a un valor `int` como un campo de dos dígitos. La **bandera 0** en el especificador de formato indica que los valores con menos dígitos que la anchura de campo (2) deben empezar con un `0` a la izquierda.

Uso de los elementos de un arreglo como contadores

En ocasiones, los programas utilizan variables tipo contador para sintetizar datos, como los resultados de una encuesta. En la figura 6.8 utilizamos contadores separados en nuestro programa para tirar dados, para rastrear el número de veces que aparecía cada una de las caras de un dado con seis lados, al tiempo que la aplicación tiraba el dado 6000 veces. En la figura 7.7 se muestra una versión de la aplicación de la figura 6.8, esta vez usando un arreglo.

```
// Fig. 7.7: TirarDado.java
// Tira un dado de seis lados 6000 veces.
import java.util.Random;

public class TirarDado
{
   public static void main( String args[] )
   {
      Random numerosAleatorios = new Random(); // generador de números aleatorios
      int frecuencia[] = new int[ 7 ]; // arreglo de contadores de frecuencia

      // tira el dado 6000 veces; usa el valor del dado como índice de frecuencia
      for ( int tiro = 1; tiro <= 6000; tiro++ )
         ++frecuencia[ 1 + numerosAleatorios.nextInt( 6 ) ];

      System.out.printf( "%s%10s\n", "Cara", "Frecuencia" );

      // imprime el valor de cada elemento del arreglo
      for ( int cara = 1; cara < frecuencia.length; cara++ )
         System.out.printf( "%4d%10d\n", cara, frecuencia[ cara ] );
   } // fin de main
} // fin de la clase TirarDado
```

```
Cara  Frecuencia
   1        1015
   2         999
   3         998
   4         996
   5        1044
   6         948
```

Figura 7.7 | Programa para tirar dados que utiliza arreglos en vez de `switch`.

La figura 7.7 utiliza el arreglo `frecuencia` (línea 10) para contar las ocurrencias de cada lado del dado. *La instrucción individual en la línea 14 de este programa sustituye las líneas 23 a 46 de la figura 6.8*. La línea 14 utiliza el valor aleatorio para determinar qué elemento de `frecuencia` debe incrementar durante cada iteración del ciclo. El cálculo en la línea 14 produce números aleatorios del 1 al 6, por lo que el arreglo `frecuencia` debe ser lo bastante grande como para poder almacenar seis contadores. Sin embargo, utilizamos un arreglo de siete elementos, en el cual ignoramos `frecuencia[ 0 ]`; es más lógico que el valor de cara 1 incremente a `frecuencia[ 1 ]` que a `frecuencia[ 0 ]`. Por ende, cada valor de cara se utiliza como subíndice para el arreglo `frecuencia`. También sustituimos las líneas 50 a 52 de la figura 6.8 por un ciclo a través del arreglo `frecuencia` para imprimir los resultados en pantalla (líneas 19 a 20).

Uso de arreglos para analizar los resultados de una encuesta

Nuestro siguiente ejemplo utiliza arreglos para sintetizar los resultados de los datos recolectados en una encuesta:

> *Se pidió a cuarenta estudiantes que calificaran la calidad de la comida en la cafetería estudiantil, en una escala del 1 al 10 (en donde 1 significa pésimo y 10 significa excelente). Coloque las 40 respuestas en un arreglo entero y sintetice los resultados de la encuesta.*

Ésta es una típica aplicación de procesamiento de arreglos (vea la figura 7.8). Deseamos resumir el número de respuestas de cada tipo (es decir, del 1 al 10). El arreglo `respuestas` (líneas 9 a 11) es un arreglo entero de 40 elementos, y contiene las respuestas de los estudiantes a la encuesta. Utilizamos un arreglo de 11 elementos llamado `frecuencia` (línea 12) para contar el número de ocurrencias de cada respuesta. Cada elemento del arreglo se utiliza como un contador para una de las respuestas de la encuesta, y se inicializa con cero de manera predeterminada. Al igual que en la figura 7.7, ignoramos `frecuencia[ 0 ]`.

El ciclo `for` en las líneas 16 y 17 recibe las respuestas del arreglo `respuestas` una a la vez, e incrementa uno de los 10 contadores en el arreglo `frecuencia` (de `frecuencia[ 1 ]` a `frecuencia[ 10 ]`). La instrucción clave en el ciclo es la línea 17, la cual incrementa el contador de `frecuencia` apropiado, dependiendo del valor de `respuestas[ respuesta ]`.

Consideraremos varias iteraciones del ciclo `for`. Cuando la variable de control `respuesta` es `0`, el valor de `respuestas[ respuesta ]` es el valor de `respuestas[ 0 ]` (es decir, `1`), por lo que el programa interpreta a `++frecuencia[ respuestas[ respuesta ] ]` como

```
++frecuencia[ 1 ]
```

con lo cual se incrementa el valor en el elemento 1 del arreglo. Para evaluar la expresión, empiece con el valor en el conjunto más interno de corchetes (`respuesta`). Una vez que conozca el valor de `respuesta` (que es el valor de la variable de control de ciclo en la línea 16), insértelo en la expresión y evalúe el siguiente conjunto más externo de corchetes (`respuestas[ respuesta ]`, que es un valor seleccionado del arreglo `respuestas` en las líneas 9 a 11). Después utilice el valor resultante como subíndice del arreglo `frecuencia`, para especificar cuál contador se incrementará.

Cuando `respuesta` es `1`, `respuestas[ respuesta ]` es el valor de `respuestas[ 1 ]` (`2`), por lo que el programa interpreta a `++frecuencia[ respuestas[ respuesta ] ]` como

```
++frecuencia[ 2 ]
```

con lo cual se incrementa el elemento `2` del arreglo.

Cuando `respuesta` es `2`, `respuestas[ respuesta ]` es el valor de `respuestas[ 2 ]` (`6`), por lo que el programa interpreta a `++frecuencia[ respuestas [ respuesta ] ]` como

```
++frecuencia[ 6 ]
```

con lo cual se incrementa el elemento `6` del arreglo, y así en lo sucesivo. Sin importar el número de respuestas procesadas en la encuesta, el programa sólo requiere un arreglo de 11 elementos (en el cual se ignora el elemento cero) para resumir los resultados, ya que todos los valores de las respuestas se encuentran entre 1 y 10, y los valores de subíndice para un arreglo de 11 elementos son del 0 al 10.

Si los datos en el arreglo `respuestas` tuvieran valores inválidos como 13, el programa trataría de sumar `1` a `frecuencia[ 13 ]`, lo cual se encuentra fuera de los límites del arreglo. Java no permite esto. Cuando se ejecuta un programa en Java, la JVM comprueba los subíndices del arreglo para asegurarse que sean válidos (es

```java
// Fig. 7.8: EncuestaEstudiantes.java
// Programa de análisis de una encuesta.

public class EncuestaEstudiantes
{
   public static void main( String args[] )
   {
      // arreglo de respuestas a una encuesta
      int respuestas[] = { 1, 2, 6, 4, 8, 5, 9, 7, 8, 10, 1, 6, 3, 8, 6,
         10, 3, 8, 2, 7, 6, 5, 7, 6, 8, 6, 7, 5, 6, 6, 5, 6, 7, 5, 6,
         4, 8, 6, 8, 10 };
      int frecuencia[] = new int[ 11 ]; // arreglo de contadores de frecuencia

      // para cada respuesta, selecciona el elemento de respuestas y usa ese valor
      // como índice de frecuencia para determinar el elemento a incrementar
      for ( int respuesta = 0; respuesta < respuestas.length; respuesta++ )
         ++frecuencia[ respuestas[ respuesta ] ];

      System.out.printf( "%s%10s\n", "Calificacion", "Frecuencia" );

      // imprime el valor de cada elemento del arreglo
      for ( int calificacion = 1; calificacion < frecuencia.length; calificacion++ )
         System.out.printf( "%6d%10d\n", calificacion, frecuencia[ calificacion ] );
   } // fin de main
} // fin de la clase EncuestaEstudiantes
```

```
Calificacion  Frecuencia
           1           2
           2           2
           3           2
           4           2
           5           5
           6          11
           7           5
           8           7
           9           1
          10           3
```

Figura 7.8 | Programa de análisis de una encuesta.

decir, deben ser mayores o iguales a 0 y menores que la longitud del arreglo). Si un programa utiliza un subíndice inválido, Java genera una excepción para indicar que se produjo un error en el programa, en tiempo de ejecución. Puede utilizarse una instrucción de control para evitar que ocurra un error tipo "fuera de los límites". Por ejemplo, la condición en una instrucción de control podría determinar si un subíndice es válido antes de permitir que se utilice en una expresión de acceso a un arreglo.



Tip para prevenir errores 7.1

Una excepción indica que ocurrió un error en un programa. A menudo el programador puede escribir código para recuperarse de una excepción y continuar con la ejecución del programa, en vez de terminarlo en forma anormal. Cuando un programa trata de acceder a un elemento fuera de los límites del arreglo, se produce una excepción `ArrayIndexOutOfBoundsException`*. En el capítulo 13 hablaremos sobre el manejo de excepciones.*



Tip para prevenir errores 7.2

Al escribir código para iterar a través de un arreglo, hay que asegurar que el subíndice del arreglo siempre sea mayor o igual a 0 y menor que la longitud del arreglo. La condición de continuación de ciclo debe evitar el acceso a elementos fuera de este rango.

7.5 Ejemplo práctico: simulación para barajar y repartir cartas

Hasta ahora, en los ejemplos en este capítulo hemos utilizado arreglos que contienen elementos de tipos primitivos. En la sección 7.2 vimos que los elementos de un arreglo pueden ser de tipos primitivos o de tipos por referencia. En esta sección utilizaremos la generación de números aleatorios y un arreglo de elementos de tipo por referencia (a saber, objetos que representan cartas de juego) para desarrollar una clase que simule los procesos de barajar y repartir cartas. Después podremos utilizar esta clase para implementar aplicaciones que ejecuten juegos específicos de cartas. Los ejercicios al final del capítulo utilizan las clases que desarrollaremos aquí para crear una aplicación simple de póquer.

Primero desarrollaremos la clase `Carta` (figura 7.9), la cual representa una carta de juego que tiene una cara (`"As"`, `"Dos"`, `"Tres"`, …, `"Joto"`, `"Qüina"`, `"Rey"`) y un palo (`"Corazones"`, `"Diamantes"`, `"Tréboles"`, `"Espadas"`). Después desarrollaremos la clase `PaqueteDeCartas` (figura 7.10), la cual crea un paquete de 52 cartas en las que cada elemento es un objeto `Carta`. Luego construiremos una aplicación de prueba (figura 7.11) para demostrar las capacidades de barajar y repartir cartas de la clase `PaqueteDeCartas`.

La clase `Carta`

La clase `Carta` (figura 7.9) contiene dos variables de instancia `String` (`cara` y `palo`) que se utilizan para almacenar referencias al valor de la cara y al valor del palo para una `Carta` específica. El constructor de la clase (líneas 10 a 14) recibe dos objetos `String` que utiliza para inicializar `cara` y `palo`. El método `toString` (líneas 17 a 20) crea un objeto `String` que consiste en la `cara` de la `carta`, el objeto `String "de"` y el `palo` de la carta. En el capítulo 6 vimos que el operador + puede utilizarse para concatenar (es decir, combinar) varios objetos `String` para formar un objeto `String` más grande. El método `toString` de `Carta` puede invocarse en forma explícita para obtener la representación de cadena de un objeto `Carta` (por ejemplo, `"As de Espadas"`). El método `toString` de un objeto se llama en forma implícita cuando el objeto se utiliza en donde se espera un objeto `String` (por ejemplo, cuando `printf` imprime en pantalla el objeto como un `String`, usando el especificador de formato `%s`, o cuando el objeto se concatena con un objeto `String` mediante el operador +). Para que ocurra este comportamiento, `toString` debe declararse con el encabezado que se muestra en la figura 7.9.

La clase `PaqueteDeCartas`

La clase `PaqueteDeCartas` (figura 7.10) declara un arreglo de variables de instancia llamado `paquete`, el cual contiene objetos `Carta` (línea 7). Al igual que las declaraciones de arreglos de tipos primitivos, la declaración de un arreglo de objetos incluye el tipo de los elementos en el arreglo, seguido del nombre de la variable del arreglo y

```
1  // Fig. 7.9: Carta.java
2  // La clase Carta representa una carta de juego.
3
4  public class Carta
5  {
6     private String cara; // cara de la carta ("As", "Dos", ...)
7     private String palo; // palo de la carta ("Corazones", "Diamantes", ...)
8
9     // el constructor de dos argumentos inicializa la cara y el palo de la carta
10    public Carta( String caraCarta, String paloCarta )
11    {
12       cara = caraCarta; // inicializa la cara de la carta
13       palo = paloCarta; // inicializa el palo de la carta
14    } // fin del constructor de Carta con dos argumentos
15
16    // devuelve representación String de Carta
17    public String toString()
18    {
19       return cara + " de " + palo;
20    } // fin del método toString
21 } // fin de la clase Carta
```

Figura 7.9 | La clase `Carta` representa una carta de juego.

```java
// Fig. 7.10: PaqueteDeCartas.java
// La clase PaqueteDeCartas representa un paquete de cartas de juego.
import java.util.Random;

public class PaqueteDeCartas
{
   private Carta paquete[]; // arreglo de objetos Carta
   private int cartaActual; // subíndice de la siguiente Carta a repartir
   private final int NUMERO_DE_CARTAS = 52; // número constante de Cartas
   private Random numerosAleatorios; // generador de números aleatorios

   // el constructor llena el paquete de Cartas
   public PaqueteDeCartas()
   {
      String caras[] = { "As", "Dos", "Tres", "Cuatro", "Cinco", "Seis",
         "Siete", "Ocho", "Nueve", "Diez", "Joto", "Quina", "Rey" };
      String palos[] = { "Corazones", "Diamantes", "Treboles", "Espadas" };

      paquete = new Carta[ NUMERO_DE_CARTAS ]; // crea arreglo de objetos Carta
      cartaActual = 0; // establece cartaActual para que la primera Carta repartida sea paquete[ 0 ]
      numerosAleatorios = new Random(); // crea generador de números aleatorios

      // llena el paquete con objetos Carta
      for ( int cuenta = 0; cuenta < paquete.length; cuenta++ )
         paquete[ cuenta ] =
            new Carta( caras[ cuenta % 13 ], palos[ cuenta / 13 ] );
   } // fin del constructor de PaqueteDeCartas

   // baraja el paquete de Cartas con algoritmo de una pasada
   public void barajar()
   {
      // después de barajar, la repartición debe empezar en paquete[ 0 ] otra vez
      cartaActual = 0; // reinicializa cartaActual

      // para cada Carta, selecciona otra Carta aleatoria y las intercambia
      for ( int primera = 0; primera < paquete.length; primera++ )
      {
         // selecciona un número aleatorio entre 0 y 51
         int segunda =  numerosAleatorios.nextInt( NUMERO_DE_CARTAS );

         // intercambia Carta actual con la Carta seleccionada al azar
         Carta temp = paquete[ primera ];
         paquete[ primera ] = paquete[ segunda ];
         paquete[ segunda ] = temp;
      } // fin de for
   } // fin de método barajar

   // reparte una Carta
   public Carta repartirCarta()
   {
      // determina si quedan Cartas por repartir
      if ( cartaActual < paquete.length )
         return paquete[ cartaActual++ ]; // devuelve la Carta actual en el arreglo
      else
         return null; // devuelve null para indicar que se repartieron todas las Cartas
   } // fin del método repartirCarta
} // fin de la clase PaqueteDeCartas
```

Figura 7.10 | La clase `PaqueteDeCartas` representa un paquete de cartas de juego, que pueden barajarse y repartirse, una a la vez.

de corchetes (por ejemplo Carta paquete[ ]). La clase PaqueteDeCartas también declara la variable de instancia entera llamada cartaActual (línea 8), que representa la siguiente Carta a repartir del arreglo paquete, y la constante con nombre NUMERO_DE_CARTAS (línea 9), que indica el número de objetos Carta en el paquete (52).

El constructor de la clase crea una instancia del arreglo paquete (línea 19) con un tamaño igual a NUMERO_DE_CARTAS. Cuando se crea por primera vez el arreglo paquete, sus elementos son null de manera predeterminada, por lo que el constructor utiliza una instrucción for (líneas 24 a 26) para llenar el arreglo paquetes con objetos Carta. La instrucción for inicializa la variable de control cuenta con 0 e itera mientras cuenta sea menor que paquete.length, lo cual hace que cuenta tome el valor de cada entero del 0 al 51 (los subíndices del arreglo paquete). Cada objeto Carta se instancia y se inicializa con dos objetos String: uno del arreglo caras (que contiene los objetos String del "As" hasta el "Rey") y uno del arreglo palos (que contiene los objetos String "Corazones", "Diamantes", "Treboles" y "Espadas"). El cálculo cuenta % 13 siempre produce un valor de 0 a 12 (los 13 subíndices del arreglo caras en las líneas 15 y 16), y el cálculo cuenta / 13 siempre produce un valor de 0 a 3 (los cuatro subíndices del arreglo palos en la línea 17). Cuando se inicializa el arreglo paquete, contiene los objetos Carta con las caras del "As" al "Rey" en orden para cada palo ("Corazones", "Diamantes", "Treboles", "Espadas").

El método barajar (líneas 30 a 46) baraja los objetos Carta en el paquete. El método itera a través de los 52 objetos Carta (subíndices 0 a 51 del arreglo). Para cada objeto Carta se elige al azar un número entre 0 y 51 para elegir otro objeto Carta. A continuación, el objeto Carta actual y el objeto Carta seleccionado al azar se intercambian en el arreglo. Este intercambio se realiza mediante las tres asignaciones en las líneas 42 a 44. La variable extra temp almacena en forma temporal uno de los dos objetos Carta que se van a intercambiar. El intercambio no se puede realizar sólo con las dos instrucciones

```
paquete[ primera ] = paquete[ segunda ];
paquete[ segunda ] = paquete[ primera ];
```

Si paquete[primera] es el "As" de "Espadas" y paquete[segunda] es la "Quina" de "Corazones", entonces después de la primera asignación, ambos elementos del arreglo contienen la "Quina" de "Corazones" y se pierde el "As" de "Espadas"; es por ello que se necesita la variable extra temp. Una vez que termina el ciclo for, los objetos Carta se ordenan al azar. Sólo se realizan 52 intercambios en una sola pasada del arreglo completo, ¡y el arreglo de objetos Carta se baraja!

El método repartirCarta (líneas 49 a 56) reparte un objeto Carta en el arreglo. Recuerde que cartaActual indica el subíndice del siguiente objeto Carta que se repartirá (es decir, la Carta en la parte superior del paquete). Por ende, la línea 52 compara cartaActual con la longitud del arreglo paquete. Si el paquete no está vacío (es decir, si cartaActual es menor a 52), la línea 53 regresa el objeto Carta "superior" y postincrementa cartaActual para prepararse para la siguiente llamada a repartirCarta; en caso contrario, se devuelve null. En el capítulo 3 vimos que null representa una "referencia a nada".

Barajar y repartir cartas

La aplicación de la figura 7.11 demuestra las capacidades de barajar y repartir cartas de la clase PaqueteDeCartas (figura 7.10). La línea 9 crea un objeto PaqueteDeCartas llamado miPaqueteDeCartas. Recuerde que el constructor PaqueteDeCartas crea el paquete con los 52 objetos Carta, en orden por palo y por cara. La línea 10 invoca el método barajar de miPaqueteDeCartas para reordenar los objetos Carta. La instrucción for en las líneas 13 a 19 reparte los 52 objetos Carta en el paquete y los imprime en cuatro columnas, cada una con 13 objetos Carta. Las líneas 16 a 18 reparten e imprimen en pantalla cuatro objetos Carta, cada uno de los cuales se obtiene mediante la invocación al método repartirCarta de miPaqueteDeCartas. Cuando printf imprime en pantalla un objeto Carta con el especificador de formato %-20s, el método toString de Carta (declarado en las líneas 17 a 20 de la figura 7.9) se invoca en forma implícita, y el resultado se imprime justificado a la izquierda, en un campo con una anchura de 20.

7.6 Instrucción for mejorada

En ejemplos anteriores demostramos cómo utilizar las instrucciones for controladas por un contador para iterar a través de los elementos en un arreglo. En esta sección presentaremos la **instrucción for mejorada**, la cual itera a través de los elementos de un arreglo o colección sin utilizar un contador (con lo cual, evita la posibilidad de "salirse" del arreglo). Esta sección habla acerca de cómo utilizar la instrucción for mejorada para iterar a tra-

```java
// Fig. 7.11: PruebaPaqueteDeCartas.java
// Aplicación para barajar y repartir cartas.

public class PruebaPaqueteDeCartas
{
   // ejecuta la aplicación
   public static void main( String args[] )
   {
      PaqueteDeCartas miPaqueteDeCartas = new PaqueteDeCartas();
      miPaqueteDeCartas.barajar(); // coloca las Cartas en orden aleatorio

      // imprime las 52 Cartas en el orden en el que se reparten
      for ( int i = 0; i < 13; i++ )
      {
         // reparte e imprime 4 Cartas
         System.out.printf( "%-20s%-20s%-20s%-20s\n",
            miPaqueteDeCartas.repartirCarta(), miPaqueteDeCartas.repartirCarta(),
            miPaqueteDeCartas.repartirCarta(), miPaqueteDeCartas.repartirCarta() );
      } // fin de for
   } // fin de main
} // fin de la clase PruebaPaqueteDeCartas
```

```
Nueve de Espadas       Joto de Corazones      Quina de Treboles      Siete de Treboles
Seis de Corazones      Seis de Treboles       Joto de Diamantes      Tres de Diamantes
Diez de Diamantes      Cinco de Diamantes     As de Treboles         Rey de Diamantes
Siete de Corazones     Cuatro de Corazones    Cuatro de Espadas      Nueve de Corazones
Dos de Espadas         Quina de Diamantes     Dos de Corazones       Quina de Corazones
As de Corazones        Cuatro de Treboles     Cinco de Espadas       Joto de Treboles
Cinco de Treboles      Siete de Diamantes     As de Espadas          Ocho de Espadas
Nueve de Treboles      Cuatro de Diamantes    Siete de Espadas       Rey de Corazones
Quina de Espadas       Dos de Diamantes       Rey de Treboles        Diez de Corazones
Cinco de Corazones     As de Diamantes        Rey de Espadas         Joto de Espadas
Ocho de Diamantes      Tres de Espadas        Ocho de Treboles       Seis de Diamantes
Nueve de Diamantes     Tres de Treboles       Diez de Treboles       Dos de Treboles
Tres de Corazones      Ocho de Corazones      Diez de Espadas        Seis de Espadas
```

Figura 7.11 | Aplicación para barajar y repartir cartas.

vés de un arreglo. En el capítulo 19, Colecciones, veremos cómo utilizar la instrucción `for` mejorada con colecciones. La sintaxis de una instrucción `for` mejorada es:

`for (` *parámetro* : *nombreArreglo* `)`
    instrucción

en donde *parámetro* tiene dos partes: un tipo y un identificador (por ejemplo, `int numero`), y *nombreArreglo* es el arreglo a través del cual se iterará. El tipo del parámetro debe concordar con el tipo de los elementos en el arreglo. Como se muestra en el siguiente ejemplo, el identificador representa valores sucesivos en el arreglo, en iteraciones sucesivas de la instrucción `for` mejorada.

La figura 7.12 utiliza la instrucción `for` mejorada (líneas 12 y 13) para calcular la suma de los enteros en un arreglo de calificaciones de estudiantes. El tipo especificado en el parámetro para el `for` mejorado es `int`, ya que `arreglo` contiene valores `int`; el ciclo selecciona un valor `int` del arreglo durante cada iteración. La instrucción `for` mejorada itera a través de valores sucesivos en el arreglo, uno por uno. El encabezado del `for` mejorado se puede leer como "para cada iteración, asignar el siguiente elemento de `arreglo` a la variable `int numero`, después ejecutar la siguiente instrucción". Por lo tanto, para cada iteración, el identificador `numero` representa un valor `int` en `arreglo`. Las líneas 12 y 13 son equivalentes a la siguiente repetición controlada por un contador que se utiliza en las líneas 12 y 13 de la figura 7.5, para totalizar los enteros en el `arreglo`:

```java
for ( int contador = 0; contador < arreglo.length; contador++ )
   total += arreglo[ contador ];
```

La instrucción `for` mejorada simplifica el código para iterar a través de un arreglo. No obstante, observe que la instrucción `for` mejorada sólo puede utilizarse para obtener elementos del arreglo; no puede utilizarse para modificar los elementos. Si su programa necesita modificar elementos, use la instrucción `for` tradicional, controlada por contador.

La instrucción `for` mejorada se puede utilizar en lugar de la instrucción `for` controlada por contador, cuando el código que itera a través de un arreglo no requiere acceso al contador que indica el subíndice del elemento actual del arreglo. Por ejemplo, para totalizar los enteros en un arreglo se requiere acceso sólo a los valores de los elementos; el subíndice de cada elemento es irrelevante. No obstante, si un programa debe utilizar un contador por alguna razón que no sea tan sólo iterar a través de un arreglo (por ejemplo, imprimir un número de subíndice al lado del valor de cada elemento del arreglo, como en los primeros ejemplos en este capítulo), use la instrucción `for` controlada por contador.

```
// Fig. 7.12: PruebaForMejorado.java
// Uso de la instrucción for mejorada para sumar el total de enteros en un arreglo.

public class PruebaForMejorado
{
   public static void main( String args[] )
   {
      int arreglo[] = { 87, 68, 94, 100, 83, 78, 85, 91, 76, 87 };
      int total = 0;

      // suma el valor de cada elemento al total
      for ( int numero : arreglo )
         total += numero;

      System.out.printf( "Total de elementos del arreglo: %d\n", total );
   } // fin de main
} // fin de la clase PruebaForMejorado
```

```
Total de elementos del arreglo: 849
```

Figura 7.12 | Uso de la instrucción `for` mejorada para sumar el total de los enteros en un arreglo.

7.7 Paso de arreglos a los métodos

Esta sección demuestra cómo pasar arreglos y elementos individuales de un arreglo como argumentos para los métodos. Al final de la sección, hablaremos acerca de cómo se pasan todos los tipos de argumentos a los métodos. Para pasar un argumento tipo arreglo a un método, se especifica el nombre del arreglo sin corchetes. Por ejemplo, si el arreglo `temperaturasPorHora` se declara como

```
double temperaturasPorHora[ ] = new double[ 24 ];
```

entonces la llamada al método

```
modificarArreglo( temperaturasPorHora );
```

pasa la referencia del arreglo `temperaturasPorHora` al método `modificarArreglo`. Todo objeto arreglo "conoce" su propia longitud (a través de su campo `length`). Por ende, cuando pasamos a un método la referencia a un objeto arreglo, no necesitamos pasar la longitud del arreglo como un argumento adicional.

Para que un método reciba una referencia a un arreglo a través de una llamada a un método, la lista de parámetros del método debe especificar un parámetro tipo arreglo. Por ejemplo, el encabezado para el método `modificarArreglo` podría escribirse así:

```
void modificarArreglo( int b[] )
```

lo cual indica que `modificarArreglo` recibe la referencia de un arreglo de enteros en el parámetro `b`. La llamada a este método pasa la referencia al arreglo `temperaturaPorHoras`, de manera que cuando el método llamado utiliza la variable `b` tipo arreglo, hace referencia al mismo objeto arreglo como `temperaturasPorHora` en el método que hizo la llamada.

Cuando un argumento para un método es todo un arreglo, o un elemento individual de un arreglo de un tipo por referencia, el método llamado recibe una copia de la referencia. Sin embargo, cuando un argumento para un método es un elemento individual de un arreglo de un tipo primitivo, el método llamado recibe una copia del valor del elemento. Dichos valores primitivos se conocen como **escalares** o **cantidades escalares**. Para pasar un elemento individual de un arreglo a un método, use el nombre indexado del elemento del arreglo como argumento en la llamada al método.

La figura 7.13 demuestra la diferencia entre pasar a un método todo un arreglo y pasar un elemento de un arreglo de tipo primitivo. La instrucción `for` mejorada en las líneas 16 y 17 imprime en pantalla los cinco elementos de `arreglo` (un arreglo de valores `int`). La línea 19 invoca al método `modificarArreglo` y le pasa `arreglo` como argumento. El método `modificarArreglo` (líneas 36 a 40) recibe una copia de la referencia a `arreglo` y utiliza esta referencia para multiplicar cada uno de los elementos de `arreglo` por 2. Para demostrar que se modificaron los elementos de `arreglo`, la instrucción `for` en las líneas 23 y 24 imprime en pantalla los cinco elementos de `arreglo` otra vez. Como se muestra en la salida, el método `modificarArreglo` duplicó el valor de cada elemento. Observe que no pudimos usar la instrucción `for` mejorada en las líneas 38 y 39, ya que estamos modificando los elementos del arreglo.

```
1  // Fig. 7.13: PasoArreglo.java
2  // Paso de arreglos y elementos individuales de un arreglo a los métodos.
3
4  public class PasoArreglo
5  {
6     // main crea el arreglo y llama a modificarArreglo y a modificarElemento
7     public static void main( String args[] )
8     {
9        int arreglo[] = { 1, 2, 3, 4, 5 };
10
11       System.out.println(
12          "Efectos de pasar una referencia a un arreglo completo:\n" +
13          "Los valores del arreglo original son:" );
14
15       // imprime los elementos originales del arreglo
16       for ( int valor : arreglo )
17          System.out.printf( "   %d", valor );
18
19       modificarArreglo( arreglo ); // pasa la referencia al arreglo
20       System.out.println( "\n\nLos valores del arreglo modificado son:" );
21
22       // imprime los elementos modificados del arreglo
23       for ( int valor : arreglo )
24          System.out.printf( "   %d", valor );
25
26       System.out.printf(
27          "\n\nEfectos de pasar el valor de un elemento del arreglo:\n" +
28          "arreglo[3] antes de modificarElemento: %d\n”, arreglo[ 3 ] );
29
30       modificarElemento( arreglo[ 3 ] ); // intento por modificar arreglo[ 3 ]
31       System.out.printf(
32          "arreglo[3] despues de modificarElemento: %d\n", arreglo[ 3 ] );
33    } // fin de main
34
35    // multiplica cada elemento de un arreglo por 2
```

Figura 7.13 | Paso de arreglos y elementos individuales de un arreglo a los métodos. (Parte 1 de 2).

```
36     public static void modificarArreglo( int arreglo2[] )
37     {
38        for ( int contador = 0; contador < arreglo2.length; contador++ )
39           arreglo2[ contador ] *= 2;
40     } // fin del método modificarArreglo
41
42     // multiplica el argumento por 2
43     public static void modificarElemento( int elemento )
44     {
45        elemento *= 2;
46        System.out.printf(
47           "Valor del elemento en modificarElemento: %d\n", elemento );
48     } // fin del método modificarElemento
49  } // fin de la clase PasoArreglo
```

```
Efectos de pasar una referencia a un arreglo completo:
Los valores del arreglo original son:
   1   2   3   4   5

Los valores del arreglo modificado son:
   2   4   6   8   10

Efectos de pasar el valor de un elemento del arreglo:
arreglo[3] antes de modificarElemento: 8
Valor del elemento en modificarElemento: 16
arreglo[3] despues de modificarElemento: 8
```

Figura 7.13 | Paso de arreglos y elementos individuales de un arreglo a los métodos. (Parte 2 de 2).

La figura 7.13 demuestra a continuación que, cuando se pasa una copia de un elemento individual de un arreglo de tipo primitivo a un método, si se modifica la copia en el método que se llamó, el valor original de ese elemento no se ve afectado en el arreglo del método que hizo la llamada. Las líneas 26 a 28 imprimen en pantalla el valor de `arreglo[ 3 ]` (8) antes de invocar al método `modificarElemento`. La línea 30 llama al método `modificarElemento` y le pasa `arreglo[ 3 ]` como argumento. Recuerde que `arreglo[ 3 ]` es en realidad un valor `int` (8) en `arreglo`. Por lo tanto, el programa pasa una copia del valor de `arreglo[ 3 ]`. El método `modificarElemento` (líneas 43 a 48) multiplica el valor recibido como argumento por 2, almacena el resultado en su parámetro `elemento` y después imprime en pantalla el valor de `elemento` (16). Como los parámetros de los métodos, al igual que las variables locales, dejan de existir cuando el método en el que se declaran termina su ejecución, el parámetro `elemento` del método se destruye cuando `modificarElemento` termina. Por lo tanto, cuando el programa devuelve el control a `main`, las líneas 31 y 32 imprimen en pantalla el valor de `arreglo[ 3 ]` que no se modificó (es decir, 8).

Notas acerca del paso de argumentos a los métodos

El ejemplo anterior demostró las distintas maneras en las que se pasan los arreglos y los elementos de arreglos de tipos primitivos a los métodos. Ahora veremos con más detalle la forma en que se pasan los argumentos a los métodos en general. En muchos lenguajes de programación, dos formas de pasar argumentos en las llamadas a métodos son el **paso por valor** y el **paso por referencia** (también conocidas como **llamada por valor** y **llamada por referencia**). Cuando se pasa un argumento por valor, se pasa una copia del valor del argumento al método que se llamó. Este método trabaja exclusivamente con la copia. Las modificaciones a la copia del método que se llamó no afectan el valor de la variable original en el método que hizo la llamada.

Cuando se pasa un argumento por referencia, el método que se llamó puede acceder al valor del argumento en el método que hizo la llamada directamente, y puede modificar esos datos si es necesario. El paso por referencia mejora el rendimiento, al eliminar la necesidad de copiar cantidades de datos posiblemente extensas.

A diferencia de otros lenguajes, Java no permite a los programadores elegir el paso por valor o el paso por referencia; todos los argumentos se pasan por valor. Una llamada a un método puede pasar dos tipos de valores:

copias de valores primitivos (como valores de tipo `int` y `double`) y copias de referencias a objetos (incluyendo las referencias a arreglos). Los objetos en sí no pueden pasarse a los métodos. Cuando un método modifica un parámetro de tipo primitivo, las modificaciones a ese parámetro no tienen efecto en el valor original del argumento en el método que hizo la llamada. Por ejemplo, cuando la línea 30 en `main` de la figura 7.13 pasa `arreglo[ 3 ]` al método `modificarElemento`, la instrucción en la línea 45 que duplica el valor del parámetro `elemento` no tiene efecto sobre el valor de `arreglo[ 3 ]` en `main`. Esto también se aplica para los parámetros de tipo por referencia. Si usted modifica un parámetro de tipo por referencia al asignarle la referencia de otro objeto, el parámetro hace referencia al nuevo objeto, pero la referencia almacenada en la variable del método que hizo la llamada sigue haciendo referencia al objeto original.

Aunque la referencia a un objeto se pasa por valor, un método puede de todas formas interactuar con el objeto al que se hace referencia, llamando a sus métodos `public` mediante el uso de la copia de la referencia al objeto. Como la referencia almacenada en el parámetro es una copia de la referencia que se pasó como argumento, el parámetro en el método que se llamó y el argumento en el método que hizo la llamada hacen referencia al mismo objeto en la memoria. Por ejemplo, en la figura 7.13, tanto el parámetro `arreglo2` en el método `modificarArreglo` como la variable `arreglo` en `main` hacen referencia al mismo objeto en la memoria. Cualquier modificación que se realice usando el parámetro `arreglo2` se lleva a cabo en el mismo objeto al que hace referencia la variable que se pasó como argumento en el método que hizo la llamada. En la figura 7.13, las modificaciones realizadas en `modificarArreglo` en las que se utiliza `arreglo2`, afectan al contenido del objeto arreglo al que hace referencia `arreglo` en `main`. De esta forma, con una referencia a un objeto, el método que se llamó puede manipular el objeto del método que hizo la llamada directamente.



Tip de rendimiento 7.1

Pasar arreglos por referencia tiene sentido por cuestiones de rendimiento. Si los arreglos se pasaran por valor, se pasaría una copia de cada elemento. En los arreglos grandes que se pasan con frecuencia, esto desperdiciaría tiempo y consumiría una cantidad considerable de almacenamiento para las copias de los arreglos.

7.8 Ejemplo práctico: la clase `LibroCalificaciones` que usa un arreglo para almacenar las calificaciones

En esta sección desarrollaremos aún más la clase `LibroCalificaciones`, que presentamos en el capítulo 3 y expandimos en los capítulos 4 y 5. Recuerde que esta clase representa un libro de calificaciones utilizado por un instructor para almacenar y analizar un conjunto de calificaciones de estudiantes. Las versiones anteriores de esta clase procesan un conjunto de calificaciones introducidas por el usuario, pero no mantienen los valores de las calificaciones individuales en variables de instancia de la clase. Por ende, los cálculos repetidos requieren que el usuario vuelva a introducir las mismas calificaciones. Una manera de resolver este problema sería almacenar cada calificación introducida por el usuario en una instancia individual de la clase. Por ejemplo, podríamos crear las variables de instancia `calificacion1`, `calificacion2`, …, `calificacion10` en la clase `LibroCalificaciones` para almacenar 10 calificaciones de estudiantes. No obstante, el código para totalizar las calificaciones y determinar el promedio de la clase sería voluminoso, y la clase no podría procesar más de 10 calificaciones a la vez. En esta sección resolvemos este problema, almacenando las calificaciones en un arreglo.

Almacenar las calificaciones de los estudiantes en un arreglo en la clase `LibroCalificaciones`

La versión de la clase `LibroCalificaciones` (figura 7.14) que presentamos aquí utiliza un arreglo de enteros para almacenar las calificaciones de varios estudiantes en un solo examen. Esto elimina la necesidad de introducir varias veces el mismo conjunto de calificaciones. El arreglo `calificaciones` se declara como una variable de instancia en la línea 7; por lo tanto, cada objeto `LibroCalificaciones` mantiene su propio conjunto de calificaciones. El constructor de la clase (líneas 10 a 14) tiene dos parámetros: el nombre del curso y un arreglo de calificaciones. Cuando una aplicación (por ejemplo, la clase `PruebaLibroCalificaciones` en la figura 7.15) crea un objeto `LibroCalificaciones`, la aplicación pasa un arreglo `int` existente al constructor, el cual asigna la referencia del arreglo a la variable de instancia `calificaciones` (línea 13). El tamaño del arreglo `calificaciones` se determina en base a la clase que pasa el arreglo al constructor. Por ende, un objeto `LibroCalificaciones` puede procesar un número de calificaciones variable. Los valores de las calificaciones en el arreglo que se pasa podría introducirlos un usuario desde el teclado, o podrían leerse desde un archivo en el disco (como veremos en el capítulo 14). En

nuestra aplicación de prueba, simplemente inicializamos un arreglo con un conjunto de valores de calificaciones (figura 7.15, línea 10). Una vez que las calificaciones se almacenan en una variable de instancia llamada `calificaciones` de la clase `LibroCalificaciones`, todos los métodos de la clase pueden acceder a los elementos de `calificaciones` según sea necesario, para realizar varios cálculos.

```
// Fig. 7.14: LibroCalificaciones.java
// Libro de calificaciones que utiliza un arreglo para almacenar las calificaciones de
una prueba.

public class LibroCalificaciones
{
   private String nombreDelCurso; // nombre del curso que representa este
   LibroCalificaciones
   private int calificaciones[]; // arreglo de calificaciones de estudiantes

   // el constructor de dos argumentos inicializa nombreDelCurso y el arreglo
   calificaciones
   public LibroCalificaciones( String nombre, int arregloCalif[] )
   {
      nombreDelCurso = nombre; // inicializa nombreDelCurso
      calificaciones = arregloCalif; // almacena las calificaciones
   } // fin del constructor de LibroCalificaciones con dos argumentos

   // método para establecer el nombre del curso
   public void establecerNombreDelCurso( String nombre )
   {
      nombreDelCurso = nombre; // almacena el nombre del curso
   } // fin del método establecerNombreDelCurso

   // método para obtener el nombre del curso
   public String obtenerNombreDelCurso()
   {
      return nombreDelCurso;
   } // fin del método obtenerNombreDelCurso

   // muestra un mensaje de bienvenida al usuario de LibroCalificaciones
   public void mostrarMensaje()
   {
      // obtenerNombreDelCurso obtiene el nombre del curso
      System.out.printf( "Bienvenido al libro de calificaciones para\n%s!\n\n",
         obtenerNombreDelCurso() );
   } // fin del método mostrarMensaje

   // realiza varias operaciones sobre los datos
   public void procesarCalificaciones()
   {
      // imprime el arreglo de calificaciones
      imprimirCalificaciones();

      // llama al método obtenerPromedio para calcular la calificación promedio
      System.out.printf( "\nEl promedio de la clase es %.2f\n", obtenerPromedio() );

      // llama a los métodos obtenerMinima y obtenerMaxima
      System.out.printf( "La calificacion mas baja es %d\nLa calificacion mas alta es
      %d\n\n",
         obtenerMinima(), obtenerMaxima() );
```

Figura 7.14 | La clase `LibroCalificaciones` que usa un arreglo para almacenar las calificaciones de una prueba. (Parte 1 de 3).

```

      // llama a imprimirGraficoBarras para imprimir el gráfico de distribución de calificaciones
         imprimirGraficoBarras();
   } // fin del método procesarCalificaciones

   // busca la calificación más baja
   public int obtenerMinima()
   {
      int califBaja = calificaciones[ 0 ]; // asume que calificaciones[ 0 ] es la más baja

      // itera a través del arreglo de calificaciones
      for ( int calificacion : calificaciones )
      {
         // si calificación es menor que califBaja, se asigna a califBaja
         if ( calificacion < califBaja )
            califBaja = calificacion; // nueva calificación más baja
      } // fin de for

      return califBaja; // devuelve la calificación más baja
   } // fin del método obtenerMinima

   // busca la calificación más alta
   public int obtenerMaxima()
   {
      int califAlta = calificaciones[ 0 ]; // asume que calificaciones[ 0 ] es la más alta

      // itera a través del arreglo de calificaciones
      for ( int calificacion : calificaciones )
      {
         // si calificacion es mayor que califAlta, se asigna a califAlta
         if ( calificacion > califAlta )
            califAlta = calificacion; // nueva calificación más alta
      } // fin de for

      return califAlta; // devuelve la calificación más alta
   } // fin del método obtenerMaxima

   // determina la calificación promedio de la prueba
   public double obtenerPromedio()
   {
      int total = 0; // inicializa el total

      // suma las calificaciones para un estudiante
      for ( int calificacion : calificaciones )
         total += calificacion;

      // devuelve el promedio de las calificaciones
      return (double) total / calificaciones.length;
   } // fin del método obtenerPromedio

   // imprime gráfico de barras que muestra la distribución de las calificaciones
   public void imprimirGraficoBarras()
   {
      System.out.println( "Distribucion de calificaciones:" );

```

Figura 7.14 | La clase LibroCalificaciones que usa un arreglo para almacenar las calificaciones de una prueba. (Parte 2 de 3).

```
103         // almacena la frecuencia de las calificaciones en cada rango de 10 calificaciones
104         int frecuencia[] = new int[ 11 ];
105 
106         // para cada calificación, incrementa la frecuencia apropiada
107         for ( int calificacion : calificaciones )
108            ++frecuencia[ calificacion / 10 ];
109 
110         // para cada frecuencia de calificación, imprime una barra en el gráfico
111         for ( int cuenta = 0; cuenta < frecuencia.length; cuenta++ )
112         {
113            // imprime etiquetas de las barras ( "00-09: ", ..., "90-99: ", "100: " )
114            if ( cuenta == 10 )
115               System.out.printf( "%5d: ", 100 );
116            else
117               System.out.printf( "%02d-%02d: ",
118                  cuenta * 10, cuenta * 10 + 9  );
119 
120            // imprime barra de asteriscos
121            for ( int estrellas = 0; estrellas < frecuencia[ cuenta ]; estrellas++ )
122               System.out.print( "*" );
123 
124            System.out.println(); // inicia una nueva línea de salida
125         } // fin de for externo
126      } // fin del método imprimirGraficoBarras
127 
128      // imprime el contenido del arreglo de calificaciones
129      public void imprimirCalificaciones()
130      {
131         System.out.println( "Las calificaciones son:\n" );
132 
133         // imprime la calificación de cada estudiante
134         for ( int estudiante = 0; estudiante < calificaciones.length; estudiante++ )
135            System.out.printf( "Estudiante %2d: %3d\n",
136               estudiante + 1, calificaciones[ estudiante ] );
137      } // fin del método imprimirCalificaciones
138   } // fin de la clase LibroCalificaciones
```

Figura 7.14 | La clase `LibroCalificaciones` que usa un arreglo para almacenar las calificaciones de una prueba. (Parte 3 de 3).

El método `procesarCalificaciones` (líneas 37 a 51) contiene una serie de llamadas a métodos que produce un reporte en el que se resumen las calificaciones. La línea 40 llama al método `imprimirCalificaciones` para imprimir el contenido del arreglo `calificaciones`. Las líneas 134 a 136 en el método `imprimirCalificaciones` utilizan una instrucción `for` para imprimir las calificaciones de los estudiantes. En este caso se debe utilizar una instrucción `for` controlada por contador, ya que las líneas 135 y 136 utilizan el valor de la variable contador `estudiante` para imprimir cada calificación enseguida de un número de estudiante específico (vea la figura 7.15). Aunque los subíndices de los arreglos empiezan en 0, lo común es que el profesor enumere a los estudiantes empezando desde 1. Por ende, las líneas 135 y 136 imprimen `estudiante + 1` como el número de estudiante para producir las etiquetas `"Estudiante 1: "`, `"Estudiante 2: "`, y así en lo sucesivo.

A continuación, el método `procesarCalificaciones` llama al método `obtenerPromedio` (línea 43) para obtener el promedio de las calificaciones en el arreglo. El método `obtenerPromedio` (líneas 86 a 96) utiliza una instrucción `for` mejorada para totalizar los valores en el arreglo `calificaciones` antes de calcular el promedio. El parámetro en el encabezado de la instrucción `for` mejorada (por ejemplo, `int calificacion`) indica que para cada iteración, la variable `int calificacion` recibe un valor en el arreglo `calificaciones`. Observe que el cálculo del promedio en la línea 95 utiliza `calificaciones.length` para determinar el número de calificaciones que se van a promediar.

Las líneas 46 y 47 en el método `procesarCalificaciones` llaman a los métodos `obtenerMinima` y `obtenerMaxima` para determinar las calificaciones más baja y más alta de cualquier estudiante en el examen, en forma respectiva. Cada uno de estos métodos utiliza una instrucción `for` mejorada para iterar a través del arreglo `calificaciones`. Las líneas 59 a 64 en el método `obtenerMinima` iteran a través del arreglo. Las líneas 62 y 63 comparan cada calificación con `califBaja`; si una calificación es menor que `califBaja`, a `califBaja` se le asigna esa calificación. Cuando la línea 66 se ejecuta, `califBaja` contiene la calificación más baja en el arreglo. El método `obtenerMaxima` (líneas 70 a 83) funciona de manera similar al método `obtenerMinima`.

Por último, la línea 50 en el método `procesarCalificaciones` llama al método `imprimirGraficoBarras` para imprimir un gráfico de distribución de los datos de las calificaciones, mediante el uso de una técnica similar a la de la figura 7.6. En ese ejemplo, calculamos en forma manual el número de calificaciones en cada categoría (es decir, de 0 a 9, de 10 a 19, …, de 90 a 99 y 100) con sólo analizar un conjunto de calificaciones. En este ejemplo, las líneas 107 y 108 utilizan una técnica similar a la de las figuras 7.7 y 7.8 para calcular la frecuencia de las calificaciones en cada categoría. La línea 104 declara y crea el arreglo `frecuencia` de 11 valores `int` para almacenar la frecuencia de las calificaciones en cada categoría de éstas. Para cada `calificacion` en el arreglo `calificaciones`, las líneas 107 y 108 incrementan el elemento apropiado del arreglo `frecuencia`. Para determinar qué elemento se debe incrementar, la línea 108 divide la `calificacion` actual entre 10, mediante la división entera. Por ejemplo, si `calificacion` es `85`, la línea 108 incrementa `frecuencia[ 8 ]` para actualizar la cuenta de calificaciones en el rango 80-89. Las líneas 111 a 125 imprimen a continuación el gráfico de barras (vea la figura 7.15), con base en los valores en el arreglo `frecuencia`. Al igual que las líneas 23 y 24 de la figura 7.6, las líneas 121 y 122 de la figura 7.14 utilizan un valor en el arreglo `frecuencia` para determinar el número de asteriscos a imprimir en cada barra.

La clase `PruebaLibroCalificaciones` *para demostrar la clase* `LibroCalificaciones`

La aplicación de la figura 7.15 crea un objeto de la clase `LibroCalificaciones` (figura 7.14) mediante el uso del arreglo `int arregloCalif` (que se declara y se inicializa en la línea 10). Las líneas 12 y 13 pasan el nombre de un curso y `arregloCalif` al constructor de `LibroCalificaciones`. La línea 14 imprime un mensaje de bienvenida, y la línea 15 invoca el método `procesarCalificaciones` del objeto `LibroCalificaciones`. La salida muestra el resumen de las 10 calificaciones en `miLibroCalificaciones`.

```
1  // Fig. 7.15: PruebaLibroCalificaciones.java
2  // Crea objeto LibroCalificaciones, usando un arreglo de calificaciones.
3
4  public class PruebaLibroCalificaciones
5  {
6     // el método main comienza la ejecución del programa
7     public static void main( String args[] )
8     {
9        // arreglo unidimensional de calificaciones de estudiantes
10       int arregloCalif[] = { 87, 68, 94, 100, 83, 78, 85, 91, 76, 87 };
11
12       LibroCalificaciones miLibroCalificaciones = new LibroCalificaciones(
13          "CS101 Introduccion a la programacion en Java", arregloCalif );
14       miLibroCalificaciones.mostrarMensaje();
15       miLibroCalificaciones.procesarCalificaciones();
16    } // fin de main
17 } // fin de la clase PruebaLibroCalificaciones
```

```
Bienvenido al libro de calificaciones para
CS101 Introduccion a la programacion en Java!
```

Figura 7.15 | `PruebaLibroCalificaciones` crea un objeto `LibroCalificaciones` usando un arreglo de calificaciones, y después invoca al método `procesarCalificaciones` para analizarlas. (Parte 1 de 2).

```
Las calificaciones son:

Estudiante  1:  87
Estudiante  2:  68
Estudiante  3:  94
Estudiante  4: 100
Estudiante  5:  83
Estudiante  6:  78
Estudiante  7:  85
Estudiante  8:  91
Estudiante  9:  76
Estudiante 10:  87

El promedio de la clase es 84.90
La calificacion mas baja es 68
La calificacion mas alta es 100

Distribucion de calificaciones:
00-09:
10-19:
20-29:
30-39:
40-49:
50-59:
60-69: *
70-79: **
80-89: ****
90-99: **
  100: *
```

Figura 7.15 | `PruebaLibroCalificaciones` crea un objeto `LibroCalificaciones` usando un arreglo de calificaciones, y después invoca al método `procesarCalificaciones` para analizarlas. (Parte 2 de 2).



Observación de ingeniería de software 7.1

Un arnés de prueba (o aplicación de prueba) es responsable de crear un objeto de la clase que se probará y de proporcionarle datos. Estos datos podrían provenir de cualquiera de varias fuentes. Los datos de prueba pueden colocarse directamente en un arreglo con un inicializador de arreglos, pueden provenir del usuario mediante el teclado, de un archivo (como veremos en el capítulo 14) o pueden provenir de una red (como veremos en el capítulo 24). Después de pasar estos datos al constructor de la clase para instanciar el objeto, este arnés de prueba debe llamar al objeto para probar sus métodos y manipular sus datos. La recopilación de datos en el arnés de prueba de esta forma permite a la clase manipular datos de varias fuentes.

7.9 Arreglos multidimensionales

Los arreglos multidimensionales de dos dimensiones se utilizan con frecuencia para representar **tablas de valores**, las cuales consisten en información ordenada en **filas** y **columnas**. Para identificar un elemento específico de una tabla, debemos especificar dos subíndices. Por convención, el primero identifica la fila del elemento y el segundo su columna. Los arreglos que requieren dos subíndices para identificar un elemento específico se llaman **arreglos bidimensionales** (los arreglos multidimensionales pueden tener más de dos dimensiones). Java no soporta los arreglos multidimensionales directamente, pero permite al programador especificar arreglos unidimensionales, cuyos elementos sean también arreglos unidimensionales, con lo cual se obtiene el mismo efecto. La figura 7.16 ilustra un arreglo bidimensional `a`, que contiene tres filas y cuatro columnas (es decir, un arreglo de tres por cuatro). En general, a un arreglo con *m* filas y *n* columnas se le llama **arreglo de *m* por *n***.

Cada elemento en el arreglo `a` se identifica en la figura 7.16 mediante una expresión de acceso a un arreglo de la forma `a [ fila ][ columna ]`; `a` es el nombre del arreglo, `fila` y `columna` son los subíndices que identifican en forma única a cada elemento en el arreglo `a` por número de fila y columna. Observe que los nombres de los

	Columna 0	Columna 1	Columna 2	Columna 3
Fila 0	a[0][0]	a[0][1]	a[0][2]	a[0][3]
Fila 1	a[1][0]	a[1][1]	a[1][2]	a[1][3]
Fila 2	a[2][0]	a[2][1]	a[2][2]	a[2][3]

Subíndice de columna
Subíndice de fila
Nombre del arreglo

Figura 7.16 | Arreglo bidimensional con tres filas y cuatro columnas.

elementos en la fila 0 tienen todos un primer subíndice de `0`, y los nombres de los elementos en la columna 3 tienen un segundo subíndice de `3`.

Arreglos de arreglos unidimensionales

Al igual que los arreglos unidimensionales, los arreglos multidimensionales pueden inicializarse mediante inicializadores de arreglos en las declaraciones. Un arreglo bidimensional `b` con dos filas y dos columnas podría declararse e inicializarse con **inicializadores de arreglos anidados**, como se muestra a continuación:

```
int b[ ] [ ] = { { 1, 2 }, {3, 4} };
```

Los valores del inicializador se agrupan por fila entre llaves. Así, `1` y `2` inicializan a `b[ 0 ][ 0 ]` y `b[ 0 ][ 1 ]`, respectivamente; `3` y `4` inicializan a `b[ 1 ][ 0 ]` y `b[ 1 ][ 1 ]`, respectivamente. El compilador cuenta el número de inicializadores de arreglos anidados (representados por conjuntos de llaves dentro de las llaves externas) en la declaración del arreglo, para determinar el número de filas en el arreglo `b`. El compilador cuenta los valores inicializadores en el inicializador de arreglos anidado de una fila, para determinar el número de columnas en esa fila. Como veremos en unos momentos, esto significa que las filas pueden tener distintas longitudes.

Los arreglos multidimensionales se mantienen como arreglos de arreglos unidimensionales. Por lo tanto, el arreglo `b` en la declaración anterior está realmente compuesto de dos arreglos unidimensionales separados: uno que contiene los valores en la primera lista inicializadora anidada `{ 1, 2 }` y uno que contiene los valores en la segunda lista inicializadora anidada `{ 3, 4 }`. Así, el arreglo `b` en sí es un arreglo de dos elementos, cada uno de los cuales es un arreglo unidimensional de valores `int`.

Arreglos bidimensionales con filas de distintas longitudes

La forma en que se representan los arreglos multidimensionales los hace bastante flexibles. De hecho, las longitudes de las filas en el arreglo `b` no tienen que ser iguales. Por ejemplo,

```
int b[ ][ ] = { { 1, 2 }, { 3, 4, 5 } };
```

crea el arreglo entero `b` con dos elementos (los cuales se determinan según el número de inicializadores de arreglos anidados) que representan las filas del arreglo bidimensional. Cada elemento de `b` es una referencia a un arreglo unidimensional de variables `int`. El arreglo `int` de la fila 0 es un arreglo unidimensional con dos elementos (`1` y `2`), y el arreglo `int` de la fila 1 es un arreglo unidimensional con tres elementos (`3`, `4` y `5`).

Creación de arreglos bidimensionales mediante expresiones de creación de arreglos

Un arreglo multidimensional con el mismo número de columnas en cada fila puede crearse mediante una expresión de creación de arreglos. Por ejemplo, en las siguientes líneas se declara el arreglo `b` y se le asigna una referencia a un arreglo de tres por cuatro:

```
int b[ ][ ] = new int[ 3 ][ 4 ];
```

En este caso, utilizamos los valores literales `3` y `4` para especificar el número de filas y columnas, respectivamente, pero esto no es obligatorio. Los programas también pueden utilizar variables para especificar las dimensiones de los arreglos, ya que `new` crea arreglos en tiempo de ejecución, no en tiempo de compilación. Al igual que con los arreglos unidimensionales, los elementos de un arreglo multidimensional se inicializan cuando se crea el objeto arreglo.

Un arreglo multidimensional, en el que cada fila tiene un número distinto de columnas, puede crearse de la siguiente manera:

```
int b[][] = new int[ 2 ][ ];   // crea 2 filas
b[ 0 ] = new int[ 5 ]; // crea 5 columnas para la fila 0
b[ 1 ] = new int[ 3 ]; // crea 3 columnas para la fila 1
```

Estas instrucciones crean un arreglo bidimensional con dos filas. La fila `0` tiene cinco columnas y la fila `1` tiene 3.

Ejemplo de arreglos bidimensionales: cómo mostrar los valores de los elementos

La figura 7.17 demuestra cómo inicializar arreglos bidimensionales con inicializadores de arreglos, y cómo utilizar ciclos `for` anidados para **recorrer** los arreglos (es decir, manipular cada uno de los elementos de cada arreglo).

El método `main` de la clase `InicArreglo` declara dos arreglos. En la declaración de `arreglo1` (línea 9) se utilizan inicializadores de arreglos anidados para inicializar la primera fila del arreglo con los valores 1, 2 y 3, y la segunda fila con los valores 4, 5 y 6. En la declaración de `arreglo2` (línea 10) se utilizan inicializadores anidados de distintas longitudes. En este caso, la primera fila se inicializa para tener dos elementos con los valores 1 y 2, respectivamente. La segunda fila se inicializa para tener un elemento con el valor 3. La tercera fila se inicializa para tener tres elementos con los valores 4, 5 y 6, respectivamente.

```
1  // Fig. 7.17: InicArreglo.java
2  // Inicialización de arreglos bidimensionales.
3
4  public class InicArreglo
5  {
6     // crea e imprime arreglos bidimensionales
7     public static void main( String args[] )
8     {
9        int arreglo1[][] = { { 1, 2, 3 }, { 4, 5, 6 } };
10       int arreglo2[][] = { { 1, 2 }, { 3 }, { 4, 5, 6 } };
11
12       System.out.println( "Los valores en arreglo1 por filas son" );
13       imprimirArreglo( arreglo1 ); // muestra arreglo1 por filas
14
15       System.out.println( "\nLos valores en arreglo2 por filas son" );
16       imprimirArreglo( arreglo2 ); // muestra arreglo2 por filas
17    } // fin de main
18
19    // imprime filas y columnas de un arreglo bidimensional
20    public static void imprimirArreglo( int arreglo[][] )
21    {
22       // itera a través de las filas del arreglo
23       for ( int fila = 0; fila < arreglo.length; fila++ )
24       {
25          // itera a través de las columnas de la fila actual
26          for ( int columna = 0; columna < arreglo[ fila ].length; columna++ )
27             System.out.printf( "%d  ", arreglo[ fila ][ columna ] );
28
29          System.out.println(); // inicia nueva línea de salida
30       } // fin de for externo
31    } // fin del método imprimirArreglo
32 } // fin de la clase InicArreglo
```

Figura 7.17 | Inicialización de arreglos bidimensionales. (Parte 1 de 2).

```
Los valores en arreglo1 por filas son
1  2  3
4  5  6

Los valores en arreglo2 por filas son
1  2
3
4  5  6
```

Figura 7.17 | Inicialización de arreglos bidimensionales. (Parte 2 de 2).

Las líneas 13 y 16 llaman al método `imprimirArreglo` (líneas 20 a 31) para imprimir los elementos de `arreglo1` y `arreglo2`, respectivamente. El método `imprimirArreglo` especifica el parámetro tipo arreglo como `int arreglo[][]` para indicar que el método recibe un arreglo bidimensional. La instrucción `for` (líneas 23 a 30) imprime las filas de un arreglo bidimensional. En la condición de continuación de ciclo de la instrucción `for` exterior, la expresión `arreglo.length` determina el número de filas en el arreglo. En la expresión `for` interior, la expresión `arreglo[ fila ].length` determina el número de columnas en la fila actual del arreglo. Esta condición permite al ciclo determinar el número exacto de columnas en cada fila.

Manipulaciones comunes en arreglos multidimensionales, realizadas mediante instrucciones `for`

En muchas manipulaciones comunes en arreglos se utilizan instrucciones `for`. Como ejemplo, la siguiente instrucción `for` asigna a todos los elementos en la fila 2 del arreglo `a`, en la figura 7.16, el valor de cero:

```
for ( int columna = 0; columna < a[ 2 ].length; columna++ )
    a[ 2 ][ columna ] = 0;
```

Especificamos la fila `2`; por lo tanto, sabemos que el primer índice siempre será `2` (`0` es la primera fila y `1` es la segunda). Este ciclo `for` varía solamente el segundo índice (es decir, el índice de la columna). Si la fila `2` del arreglo `a` contiene cuatro elementos, entonces la instrucción `for` anterior es equivalente a las siguientes instrucciones de asignación:

```
a[ 2 ][ 0 ] = 0;
a[ 2 ][ 1 ] = 0;
a[ 2 ][ 2 ] = 0;
a[ 2 ][ 3 ] = 0;
```

La siguiente instrucción `for` anidada suma el total de los valores de todos los elementos del arreglo `a`:

```
int total = 0;

for ( int fila = 0; fila < a.length; fila++ )
{
    for ( int columna = 0; columna < a[ fila ].length; columna++ )
        total += a[ fila ][ columna ];
}  // fin de for exterior
```

Estas instrucciones `for` anidadas suman el total de los elementos del arreglo, una fila a la vez. La instrucción `for` exterior empieza asignando `0` al índice `fila`, de manera que los elementos de la primera fila puedan totalizarse mediante la instrucción `for` interior. Después, la instrucción `for` exterior incrementa `fila` a `1`, de manera que la segunda fila pueda totalizarse. Luego, la instrucción `for` exterior incrementa `fila` a `2`, para que la tercera fila pueda totalizarse. La variable `total` puede mostrarse al terminar la instrucción `for` exterior. En el siguiente ejemplo le mostraremos cómo procesar un arreglo bidimensional de una manera similar, usando instrucciones `for` mejoradas anidadas.

7.10 Ejemplo práctico: la clase `LibroCalificaciones` que usa un arreglo bidimensional

En la sección 7.8 presentamos la clase `LibroCalificaciones` (figura 7.14), la cual utilizó un arreglo unidimensional para almacenar las calificaciones de los estudiantes en un solo examen. En la mayoría de los cursos, los estudiantes presentan varios exámenes. Es probable que los profesores quieran analizar las calificaciones a lo largo de todo el curso, tanto para un solo estudiante como para la clase en general.

Cómo almacenar las calificaciones de los estudiantes en un arreglo bidimensional en la clase ***`LibroCalificaciones`***

La figura 7.18 contiene una versión de la clase `LibroCalificaciones` que utiliza un arreglo bidimensional llamado `calificaciones`, para almacenar las calificaciones de un número de estudiantes en varios exámenes. Cada fila del arreglo representa las calificaciones de un solo estudiante durante todo el curso, y cada columna representa las calificaciones para la clase completa en uno de los exámenes que presentaron los estudiantes durante el curso. Una aplicación como `PruebaLibroCalificaciones` (figura 7.19) pasa el arreglo como argumento para el constructor de `LibroCalificaciones`. En este ejemplo, utilizamos un arreglo de diez por tres que contiene diez calificaciones de los estudiantes en tres exámenes. Cinco métodos realizan manipulaciones de arreglos para procesar las calificaciones. Cada método es similar a su contraparte en la versión anterior de la clase `LibroCalificaciones` con un arreglo unidimensional (figura 7.14). El método `obtenerMinima` (líneas 52 a 70) determina la calificación más baja de cualquier estudiante durante el semestre. El método `obtenerMaxima` (líneas 73 a 91) determina la calificación más alta de cualquier estudiante durante el semestre. El método `obtenerPromedio` (líneas 94 a 104) determina el promedio semestral de un estudiante específico. El método `imprimirGraficoBarras` (líneas 107 a 137) imprime un gráfico de barras de la distribución de todas las calificaciones de los estudiantes durante el semestre. El método `imprimirCalificaciones` (líneas 140 a 164) imprime el arreglo bidimensional en formato tabular, junto con el promedio semestral de cada estudiante.

Cada uno de los métodos `obtenerMinima`, `obtenerMaxima` e `imprimirGraficoBarras` itera a través del arreglo `calificaciones` mediante el uso de instrucciones `for` anidadas; por ejemplo, la instrucción `for` mejorada anidada de la declaración del método `obtenerMinima` (líneas 58 a 67). La instrucción `for` mejorada exterior itera a través del arreglo bidimensional `calificaciones`, asignando filas sucesivas al parámetro `califEstudiantes` en las iteraciones sucesivas. Los corchetes que van después del nombre del parámetro indican que `califEstudiantes` se refiere a un arreglo `int` unidimensional: a saber, una fila en el arreglo `calificaciones`, que contiene las calificaciones de un estudiante. Para buscar la calificación más baja en general, la instrucción `for` interior compara los elementos del arreglo unidimensional actual `califEstudiantes` a la variable `califBaja`. Por ejemplo, en la primera iteración del `for` exterior, la fila 0 de `calificaciones` se asigna al parámetro `califEstudiantes`. Después, la instrucción `for` mejorada interior itera a través de `califEstudiantes` y compara cada valor de `calificacion` con `califBaja`. Si una calificación es menor que `califBaja`, a `califBaja` se le asigna esa calificación. En la segunda iteración de la instrucción `for` mejorada exterior, la fila 1 de `calificaciones` se asigna a `califEstudiantes`, y los elementos de esta fila se comparan con la variable `califBaja`. Esto se repite hasta que se hayan recorrido todas las filas de `calificaciones`. Cuando se completa la ejecución de la instrucción anidada, `califBaja` contiene la calificación más baja en el arreglo bidimensional. El método `obtenerMaxima` trabaja en forma similar al método `obtenerMinima`.

El método `imprimirGraficoBarras` en la figura 7.18 es casi idéntico al de la figura 7.14. Sin embargo, para imprimir la distribución de calificaciones en general durante todo un semestre, el método aquí utiliza una instrucción `for` mejorada anidada (líneas 115 a 119) para crear el arreglo unidimensional `frecuencia`, con base en todas las calificaciones en el arreglo bidimensional. El resto del código en cada uno de los dos métodos `imprimirGraficoBarras` que muestran el gráfico es idéntico.

El método `imprimirCalificaciones` (líneas 140 a 164) utiliza instrucciones `for` anidadas para imprimir valores del arreglo `calificaciones`, además del promedio semestral de cada estudiante. La salida en la figura 7.19 muestra el resultado, el cual se asemeja al formato tabular del libro de calificaciones real de un profesor. Las líneas 146 y 147 imprimen los encabezados de columna para cada prueba. Aquí utilizamos una instrucción `for` controlada por contador, para poder identificar cada prueba con un número. De manera similar, la instrucción `for` en las líneas 152 a 163 imprime primero una etiqueta de fila mediante el uso de una variable contador para identificar a cada estudiante (línea 154). Aunque los subíndices de los arreglos empiezan en 0, observe que las líneas 147 y 154 imprimen `prueba + 1` y `estudiante + 1` en forma respectiva, para producir números de

```java
// Fig. 7.18: LibroCalificaciones.java
// Libro de calificaciones que utiliza un arreglo bidimensional para almacenar calificaciones.

public class LibroCalificaciones
{
   private String nombreDelCurso; // nombre del curso que representa este LibroCalificaciones
   private int calificaciones[][]; // arreglo bidimensional de calificaciones de estudiantes

   // el constructor con dos argumentos inicializa nombreDelCurso y el arreglo calificaciones
   public LibroCalificaciones( String nombre, int arregloCalif[][] )
   {
      nombreDelCurso = nombre; // inicializa nombreDelCurso
      calificaciones = arregloCalif; // almacena calificaciones
   } // fin del constructor de LibroCalificaciones con dos argumentos

   // método para establecer el nombre del curso
   public void establecerNombreDelCurso( String nombre )
   {
      nombreDelCurso = nombre; // almacena el nombre del curso
   } // fin del método establecerNombreDelCurso

   // método para obtener el nombre del curso
   public String obtenerNombreDelCurso()
   {
      return nombreDelCurso;
   } // fin del método obtenerNombreDelCurso

   // muestra un mensaje de bienvenida al usuario de LibroCalificaciones
   public void mostrarMensaje()
   {
      // obtenerNombreDelCurso obtiene el nombre del curso
      System.out.printf( "Bienvenido al libro de calificaciones para\n%s!\n\n",
         obtenerNombreDelCurso() );
   } // fin del método mostrarMensaje

   // realiza varias operaciones sobre los datos
   public void procesarCalificaciones()
   {
      // imprime el arreglo de calificaciones
      imprimirCalificaciones();

      // llama a los métodos obtenerMinima y obtenerMaxima
      System.out.printf( "\n%s %d\n%s %d\n\n ",
         "La calificación mas baja en el libro de calificaciones es", obtenerMinima(),
         "La calificación mas alta en el libro de calificaciones es", obtenerMinima() );

      // imprime gráfico de distribución de calificaciones para todas las pruebas
      imprimirGraficoBarras();
   } // fin del método procesarCalificaciones

   // busca la calificación más baja
   public int obtenerMinima()
   {
      // asume que el primer elemento del arreglo calificaciones es el más bajo
```

Figura 7.18 | Clase LibroCalificaciones que utiliza un arreglo bidimensional para almacenar calificaciones. (Parte 1 de 3).

```
      int califBaja = calificaciones[ 0 ][ 0 ];

      // itera a través de las filas del arreglo calificaciones
      for ( int califEstudiantes[] : calificaciones )
      {
         // itera a través de las columnas de la fila actual
         for ( int calificacion : califEstudiantes )
         {
            // si la calificación es menor que califBaja, la asigna a califBaja
            if ( calificacion < califBaja )
               califBaja = calificacion;
         } // fin de for interior
      } // fin de for exterior

      return califBaja; // devuelve calificación más baja
   } // fin del método obtenerMinima

   // busca la calificación más alta
   public int obtenerMaxima()
   {
      // asume que el primer elemento del arreglo calificaciones es el más alto
      int califAlta = calificaciones[ 0 ][ 0 ];

      // itera a través de las filas del arreglo calificaciones
      for ( int califEstudiantes[] : calificaciones )
      {
         // itera a través de las columnas de la fila actual
         for ( int calificacion : califEstudiantes )
         {
            // si la calificación es mayor que califAlta, la asigna a califAlta
            if ( calificacion > califAlta )
               califAlta = calificacion;
         } // fin de for interior
      } // fin de for exterior

      return califAlta; // devuelve la calificación más alta
   } // fin del método obtenerMaxima

   // determina la calificación promedio para un estudiante específico (o conjunto de
   calificaciones)
   public double obtenerPromedio( int conjuntoDeCalif[] )
   {
      int total = 0; // inicializa el total

      // suma las calificaciones para un estudiante
      for ( int calificacion : conjuntoDeCalif )
         total += calificacion;

      // devuelve el promedio de calificaciones
      return (double) total / conjuntoDeCalif.length;
   } // fin del método obtenerPromedio

   // imprime gráfico de barras que muestra la distribución de calificaciones en general
   public void imprimirGraficoBarras()
   {
      System.out.println( "Distribucion de calificaciones en general:" );

      // almacena la frecuencia de las calificaciones en cada rango de 10 calificaciones
```

Figura 7.18 | Clase `LibroCalificaciones` que utiliza un arreglo bidimensional para almacenar calificaciones. (Parte 2 de 3).

```
      int frecuencia[] = new int[ 11 ];

      // para cada calificación en LibroCalificaciones, incrementa la frecuencia apropiada
      for ( int califEstudiantes[] : calificaciones )
      {
         for ( int calificacion : califEstudiantes )
             ++frecuencia[ calificacion / 10 ];
      } // fin de for exterior

      // para cada frecuencia de calificaciones, imprime una barra en el gráfico
      for ( int cuenta = 0; cuenta < frecuencia.length; cuenta++ )
      {
         // imprime etiquetas de las barras ( "00-09: ", ..., "90-99: ", "100: " )
         if ( cuenta == 10 )
            System.out.printf( "%5d: ", 100 );
         else
            System.out.printf( "%02d-%02d: ",
              cuenta * 10, cuenta * 10 + 9  );

         // imprime barra de asteriscos
         for ( int estrellas = 0; estrellas < frecuencia[ cuenta ]; estrellas++ )
            System.out.print( "*" );

         System.out.println(); // inicia una nueva línea de salida
      } // fin de for exterior
   } // fin del método imprimirGraficoBarras

   // imprime el contenido del arreglo calificaciones
   public void imprimirCalificaciones()
   {
      System.out.println( "Las calificaciones son:\n" );
      System.out.print( "            " ); // alinea encabezados de columnas

      // crea un encabezado de columna para cada una de las pruebas
      for ( int prueba = 0; prueba < calificaciones[ 0 ].length; prueba++ )
         System.out.printf( "Prueba %d  ", prueba + 1 );

      System.out.println( "Promedio" ); // encabezado de columna de promedio de estudiantes

      // crea filas/columnas de texto que representan el arreglo calificaciones
      for ( int estudiante = 0; estudiante < calificaciones.length; estudiante++ )
      {
         System.out.printf( "Estudiante %2d", estudiante + 1 );

         for ( int prueba : calificaciones[ estudiante ] ) // imprime calificaciones de estudiante
             System.out.printf( "%8d", prueba );

         // llama al método obtenerPromedio para calcular la calificación promedio del estudiante;
         // pasa fila de calificaciones como argumento para obtenerPromedio
         double promedio = obtenerPromedio( calificaciones[ estudiante ] );
         System.out.printf( "%9.2f\n", promedio );
      } // fin de for exterior
   } // fin del método imprimirCalificaciones
} // fin de la clase LibroCalificaciones
```

Figura 7.18 | Clase LibroCalificaciones que utiliza un arreglo bidimensional para almacenar calificaciones. (Parte 3 de 3).

prueba y estudiante que empiecen en 1 (vea la figura 7.19) La instrucción `for` interna en las líneas 156 y 157 utiliza la variable contador `estudiante` de la instrucción `for` exterior para iterar a través de una fila específica del arreglo `calificaciones`, e imprime la calificación de la prueba de cada estudiante. Observe que una instrucción `for` mejorada puede anidarse en una instrucción `for` controlada por contador, y viceversa. Por último, la línea 161 obtiene el promedio semestral de cada estudiante, para lo cual pasa la fila actual de `calificaciones` (es decir, `calificaciones[ estudiante ]`) al método `obtenerPromedio`.

El método `obtenerPromedio` (líneas 94 a 104) recibe un argumento: un arreglo unidimensional de resultados de la prueba para un estudiante específico. Cuando la línea 161 llama a `obtenerPromedio`, el argumento es `calificaciones[ estudiante ]`, el cual especifica que debe pasarse una fila específica del arreglo bidimensional `calificaciones` a `obtenerPromedio`. Por ejemplo, con base en el arreglo creado en la figura 7.19, el argumento `calificaciones[ 1 ]` representa los tres valores (un arreglo unidimensional de calificaciones) almacenados en la fila `1` del arreglo bidimensional `calificaciones`. Recuerde que un arreglo bidimensional es un arreglo cuyos elementos son arreglos unidimensionales. El método `obtenerPromedio` calcula la suma de los elementos del arreglo, divide el total entre el número de resultados de la prueba y devuelve el resultado de punto flotante como un valor `double` (línea 103).

La clase `PruebaLibroCalificaciones` *que demuestra la clase* `LibroCalificaciones`

La aplicación en la figura 7.19 crea un objeto de la clase `LibroCalificaciones` (figura 7.18) mediante el uso del arreglo bidimensional de valores `int` llamado `arregloCalif` (el cual se declara e inicializa en las líneas 10 a 19). Las líneas 21 y 22 pasan el nombre de un curso y `arregloCalif` al constructor de `LibroCalificaciones`. Después, las líneas 23 y 24 invocan a los métodos `mostrarMensaje` y `procesarCalificaciones` de `miLibroCalificaciones`, para mostrar un mensaje de bienvenida y obtener un informe que sintetice las calificaciones de los estudiantes para el semestre, respectivamente.

```
1  // Fig. 7.19: PruebaLibroCalificaciones.java
2  // Crea objeto LibroCalificaciones, usando un arreglo bidimensional de calificaciones.
3
4  public class PruebaLibroCalificaciones
5  {
6     // el método main comienza la ejecución del programa
7     public static void main( String args[] )
8     {
9        // arreglo bidimensional de calificaciones de estudiantes
10       int arregloCalif[][] = { { 87, 96, 70 },
11                                { 68, 87, 90 },
12                                { 94, 100, 90 },
13                                { 100, 81, 82 },
14                                { 83, 65, 85 },
15                                { 78, 87, 65 },
16                                { 85, 75, 83 },
17                                { 91, 94, 100 },
18                                { 76, 72, 84 },
19                                { 87, 93, 73 } };
20
21       LibroCalificaciones miLibroCalificaciones = new LibroCalificaciones(
22          "CS101 Introduccion a la programacion en Java", arregloCalif );
23       miLibroCalificaciones.mostrarMensaje();
24       miLibroCalificaciones.procesarCalificaciones();
25    } // fin de main
26 } // fin de la clase PruebaLibroCalificaciones
```

Figura 7.19 | Crea un objeto `LibroCalificaciones` usando un arreglo bidimensional de calificaciones; después invoca al método `procesarCalificaciones` para analizarlas. (Parte 1 de 2).

```
Bienvenido al libro de calificaciones para
CS101 Introduccion a la programacion en Java!

Las calificaciones son:

                Prueba 1  Prueba 2   Prueba 3  Promedio
Estudiante  1        87        96         70     84.33
Estudiante  2        68        87         90     81.67
Estudiante  3        94       100         90     94.67
Estudiante  4       100        81         82     87.67
Estudiante  5        83        65         85     77.67
Estudiante  6        78        87         65     76.67
Estudiante  7        85        75         83     81.00
Estudiante  8        91        94        100     95.00
Estudiante  9        76        72         84     77.33
Estudiante 10        87        93         73     84.33

La calificacion mas baja en el libro de calificaciones es 65
La calificacion mas alta en el libro de calificaciones es 100

Distribucion de calificaciones en general:
00-09:
10-19:
20-29:
30-39:
40-49:
50-59:
60-69: ***
70-79: ******
80-89: ***********
90-99: *******
  100: ***
```

Figura 7.19 | Crea un objeto `LibroCalificaciones` usando un arreglo bidimensional de calificaciones; después invoca al método `procesarCalificaciones` para analizarlas. (Parte 2 de 2).

7.11 Listas de argumentos de longitud variable

Con las **listas de argumentos de longitud variable** podemos crear métodos que reciben un número arbitrario de argumentos. Un tipo de argumento que va precedido por una **elipsis (...) en la lista de parámetros de un método** indica que éste recibe un número variable de argumentos de ese tipo específico. Este uso de la elipsis puede ocurrir sólo una vez en una lista de parámetros, y la elipsis, junto con su tipo, debe colocarse al final de la lista. Aunque los programadores pueden utilizar la sobrecarga de métodos y el paso de arreglos para realizar gran parte de lo que se logra con los "varargs" (listas de argumentos de longitud variable), es más conciso utilizar una elipsis en la lista de parámetros de un método.

La figura 7.20 demuestra el método `promedio` (líneas 7 a 16), el cual recibe una secuencia de longitud variable de valores `double`. Java trata a la lista de argumentos de longitud variable como un arreglo cuyos elementos son del mismo tipo. Así, el cuerpo del método puede manipular el parámetro `numeros` como un arreglo de valores `double`. Las líneas 12 y 13 utilizan el ciclo `for` mejorado para recorrer el arreglo y calcular el total de los valores `double` en el arreglo. La línea 15 accede a `numeros.length` para obtener el tamaño del arreglo `numeros` y usarlo en el cálculo del promedio. Las líneas 29, 31 y 33 en `main` llaman al método `promedio` con dos, tres y cuatro argumentos, respectivamente. El método `promedio` tiene una lista de argumentos de longitud variable (línea 7), por lo que puede promediar todos los argumentos `double` que le pase el método que hace la llamada. La salida revela que cada llamada al método `promedio` devuelve el valor correcto.



Error común de programación 7.6

Colocar una elipsis en medio de una lista de parámetros de un método indicando una lista de argumentos de longitud variable es un error de sintaxis. La elipsis sólo debe colocarse al final de la lista de parámetros.

```java
// Fig. 7.20: PruebaVarargs.java
// Uso de listas de argumentos de longitud variable.

public class PruebaVarargs
{
   // calcula el promedio
   public static double promedio( double... numeros )
   {
      double total = 0.0; // inicializa el total

      // calcula el total usando la instrucción for mejorada
      for ( double d : numeros )
         total += d;

      return total / numeros.length;
   } // fin del método promedio

   public static void main( String args[] )
   {
      double d1 = 10.0;
      double d2 = 20.0;
      double d3 = 30.0;
      double d4 = 40.0;

      System.out.printf( "d1 = %.1f\nd2 = %.1f\nd3 = %.1f\nd4 = %.1f\n\n",
         d1, d2, d3, d4 );

      System.out.printf( "El promedio de d1 y d2 es %.1f\n",
         promedio( d1, d2 ) );
      System.out.printf( "El promedio de d1, d2 y d3 es %.1f\n",
         promedio( d1, d2, d3 ) );
      System.out.printf( "El promedio de d1, d2, d3 y d4 es %.1f\n",
         promedio( d1, d2, d3, d4 ) );
   } // fin de main
} // fin de la clase PruebaVarargs
```

```
d1 = 10.0
d2 = 20.0
d3 = 30.0
d4 = 40.0

El promedio de d1 y d2 es 15.0
El promedio de d1, d2 y d3 es 20.0
El promedio de d1, d2, d3 y d4 es 25.0
```

Figura 7.20 | Uso de listas de argumentos de longitud variable.

7.12 Uso de argumentos de línea de comandos

En muchos sistemas, es posible pasar argumentos desde la línea de comandos (a éstos se les conoce como **argumentos de línea de comandos**) a una aplicación, para lo cual se incluye un parámetro de tipo `String[ ]` (es decir, un arreglo de objetos `String`) en la lista de parámetros de `main`, justo igual que como hemos hecho en todas las aplicaciones de este libro. Por convención, a este parámetro se le llama `args`. Cuando se ejecuta una aplicación usando el comando `java`, Java pasa los argumentos de línea de comandos que aparecen después del nombre de la clase en el comando `java` al método `main` de la aplicación, en forma de objetos `String` en el arreglo `args`. El número de argumentos que se pasan desde la línea de comandos se obtiene accediendo al atributo `length` del arreglo. Por ejemplo, el comando `"java miClase a b"` pasa dos argumentos de línea de comandos, `a` y `b`, a la aplicación `miClase`. Observe que los argumentos de la línea de comandos se separan por espacio en blanco, no

por comas. Cuando se ejecuta este comando, el método `main` de `MiClase` recibe el arreglo `args` de dos elementos (es decir, el valor del atributo `length` de `args` es 2), en el cual `args[ 0 ]` contiene el objeto `String` `"a"` y `args[ 1 ]` contiene el objeto `String` `"b"`. Los usos comunes de los argumentos de línea de comandos incluyen pasar opciones y nombres de archivos a las aplicaciones.

La figura 7.21 utiliza tres argumentos de línea de comandos para inicializar un arreglo. Cuando se ejecuta el programa, si `args.length` no es `3`, el programa imprime un mensaje de error y termina (líneas 9 a 12). En cualquier otro caso, las líneas 14 a 32 inicializan y muestran el arreglo, con base en los valores de los argumentos de línea de comandos.

Los argumentos de línea de comandos están disponibles para `main` como objetos `String` en `args`. La línea 16 obtiene `args[ 0 ]` (un objeto `String` que especifica el tamaño del arreglo) y lo convierte en un valor `int`, que el programa utiliza para crear el arreglo en la línea 17. El método `static parseInt` de la clase `Integer` convierte su argumento `String` en un `int`.

Las líneas 20 a 21 convierten los argumentos de línea de comandos `args[ 1 ]` y `args[ 2 ]` en valores `int`, y los almacenan en `valorInicial` e `incremento`, respectivamente. Las líneas 24 y 25 calculan el valor para cada elemento del arreglo.

Los resultados de la primera ejecución muestran que la aplicación recibió un número insuficiente de argumentos de línea de comandos. La segunda ejecución utiliza los argumentos de línea de comandos 5, 0 y 4 para especificar el tamaño del arreglo (5), el valor del primer elemento (0) y el incremento de cada valor en el arreglo (4), respectivamente. Los resultados correspondientes muestran que estos valores crean un arreglo que contiene los enteros 0, 4, 8, 12 y 16. Los resultados de la tercera ejecución muestran que los argumentos de línea de comandos 10, 1 y 2 producen un arreglo cuyos 10 elementos son los enteros impares positivos del 1 al 19.

```
1  // Fig. 7.21: InicArreglo.java
2  // Uso de los argumentos de línea de comandos para inicializar un arreglo.
3
4  public class InicArreglo
5  {
6     public static void main( String args[] )
7     {
8        // comprueba el número de argumentos de línea de comandos
9        if ( args.length != 3 )
10          System.out.println(
11             "Error: Vuelva a escribir el comando completo, incluyendo\n" +
12             "el tamanio del arreglo, el valor inicial y el incremento." );
13       else
14       {
15          // obtiene el tamaño del arreglo del primer argumento de línea de comandos
16          int longitudArreglo = Integer.parseInt( args[ 0 ] );
17          int arreglo[] = new int[ longitudArreglo ]; // crea el arreglo
18
19          // obtiene el valor inicial y el incremento de los argumentos de línea de
            comandos
20          int valorInicial = Integer.parseInt( args[ 1 ] );
21          int incremento = Integer.parseInt( args[ 2 ] );
22
23          // calcula el valor para cada elemento del arreglo
24          for ( int contador = 0; contador < arreglo.length; contador++ )
25             arreglo[ contador ] = valorInicial + incremento * contador;
26
27          System.out.printf( "%s%8s\n", "Indice", "Valor" );
28
29          // muestra el índice y el valor del arreglo
30          for ( int contador = 0; contador < arreglo.length; contador++ )
31             System.out.printf( "%5d%8d\n", contador, arreglo[ contador ] );
```

Figura 7.21 | Inicialización de un arreglo, usando argumentos de línea de comandos. (Parte 1 de 2).

```
        } // fin de else
    } // fin de main
} // fin de la clase InicArreglo
```

```
java InicArreglo
Error: Vuelva a escribir el comando completo, incluyendo
el tamanio del arreglo, el valor inicial y el incremento.
```

```
java InicArreglo 5 0 4
Indice   Valor
     0       0
     1       4
     2       8
     3      12
     4      16
```

```
java InicArreglo 10 1 2
Indice   Valor
     0       1
     1       3
     2       5
     3       7
     4       9
     5      11
     6      13
     7      15
     8      17
     9      19
```

Figura 7.21 | Inicialización de un arreglo, usando argumentos de línea de comandos. (Parte 2 de 2).

7.13 (Opcional) Ejemplo práctico de GUI y gráficos: cómo dibujar arcos

Mediante el uso de las herramientas para gráficos de Java, podemos crear dibujos complejos que, si los codificáramos línea por línea, sería un proceso tedioso. En las figuras 7.22 y 7.23 utilizamos arreglos e instrucciones de repetición para dibujar un arco iris, mediante el uso del método `fillArc` de `Graphics`. El proceso de dibujar arcos en Java es similar a dibujar óvalos; un arco es simplemente una sección de un óvalo.

La figura 7.22 empieza con las instrucciones `import` usuales para ciertos dibujos (líneas 3 a 5). Las líneas 9 y 10 declaran y crean dos nuevos colores: `VIOLETA` e `INDIGO`. Como tal vez lo sepa, los colores de un arco iris son rojo, naranja, amarillo, verde, azul, índigo y violeta. Java tiene constantes predefinidas sólo para los primeros cinco colores. Las líneas 15 a 17 inicializan un arreglo con los colores del arco iris, empezando con los arcos más interiores primero. El arreglo empieza con dos elementos `Color.WHITE`, que como veremos pronto, son para dibujar los arcos vacíos en el centro del arco iris. Observe que las variables de instancia se pueden inicializar al momento de declararse, como se muestra en las líneas 10 a 17. El constructor (líneas 20 a 23) contiene una sola instrucción que llama al método **setBackground** (heredado de la clase `JPanel`) con el parámetro `Color.WHITE`. El método `setBackground` recibe un solo argumento `Color` y establece el color de fondo del componente a ese color.

La línea 30 en `paintComponent` declara la variable local `radio`, que determina el radio de cada arco. Las variables locales `centroX` y `centroY` (líneas 33 y 34) determinan la ubicación del punto medio en la base del arco iris. El ciclo en las líneas 37 a 46 utiliza la variable de control `contador` para contar en forma regresiva, partiendo del final del arreglo, dibujando los arcos más grandes primero y colocando cada arco más pequeño encima del anterior. La línea 40 establece el color para dibujar el arco actual del arreglo. La razón por la que tenemos entradas `Color.WHITE` al principio del arreglo es para crear el arco vacío en el centro. De no ser así, el centro del arco iris

```
1  // Fig. 7.22: DibujoArcoIris.java
2  // Demuestra el uso de colores en un arreglo.
3  import java.awt.Color;
4  import java.awt.Graphics;
5  import javax.swing.JPanel;
6
7  public class DibujoArcoIris extends JPanel
8  {
9     // Define los colores índigo y violeta
10    final Color VIOLETA = new Color( 128, 0, 128 );
11    final Color INDIGO = new Color( 75, 0, 130 );
12
13    // los colores a usar en el arco iris, empezando desde los más interiores
14    // Las dos entradas de color blanco producen un arco vacío en el centro
15    private Color colores[] =
16       { Color.WHITE, Color.WHITE, VIOLETA, INDIGO, Color.BLUE,
17         Color.GREEN, Color.YELLOW, Color.ORANGE, Color.RED };
18
19    // constructor
20    public DibujoArcoIris()
21    {
22       setBackground( Color.WHITE ); // establece el fondo al color blanco
23    } // fin del constructor de DibujoArcoIris
24
25    // dibuja un arco iris, usando círculos concéntricos
26    public void paintComponent( Graphics g )
27    {
28       super.paintComponent( g );
29
30       int radio = 20; // el radio de un arco
31
32       // dibuja el arco iris cerca de la parte central inferior
33       int centroX = getWidth() / 2;
34       int centroY = getHeight() - 10;
35
36       // dibuja arcos rellenos, empezando con el más exterior
37       for ( int contador = colores.length; contador > 0; contador-- )
38       {
39          // establece el color para el arco actual
40          g.setColor( colores[ contador - 1 ] );
41
42          // rellena el arco desde 0 hasta 180 grados
43          g.fillArc( centroX - contador * radio,
44             centroY - contador * radio,
45             contador * radio * 2, contador * radio * 2, 0, 180 );
46       } // fin de for
47    } // fin del método paintComponent
48 } // fin de la clase DibujoArcoIris
```

Figura 7.22 | Dibujo de un arco iris, usando arcos y un arreglo de colores.

sería un semicírculo sólido color violeta. [*Nota:* puede cambiar los colores individuales y el número de entradas en el arreglo para crear nuevos diseños].

La llamada al método `fillArc` en las líneas 43 a 45 dibuja un semicírculo relleno. El método `fillArc` requiere seis parámetros. Los primeros cuatro representan el rectángulo delimitador en el cual se dibujará el arco. Los primeros dos de estos cuatro especifican las coordenadas para la esquina superior izquierda del rectángulo delimitador, y los siguientes dos especifican su anchura y su altura. El quinto parámetro es el ángulo inicial en el óvalo, y el sexto especifica el **barrido** o la cantidad de arco que se cubrirá. El ángulo inicial y el barrido se miden en

```java
// Fig. 7.23: DrawRainbowTest.java
// Aplicación de prueba para mostrar un arco iris.
import javax.swing.JFrame;

public class PruebaDibujoArcoIris
{
   public static void main( String args[] )
   {
      DibujoArcoIris panel = new DibujoArcoIris();
      JFrame aplicacion = new JFrame();

      aplicacion.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
      aplicacion.add( panel );
      aplicacion.setSize( 400, 250 );
      aplicacion.setVisible( true );
   } // fin de main
} // fin de la clase PruebaDibujoArcoIris
```

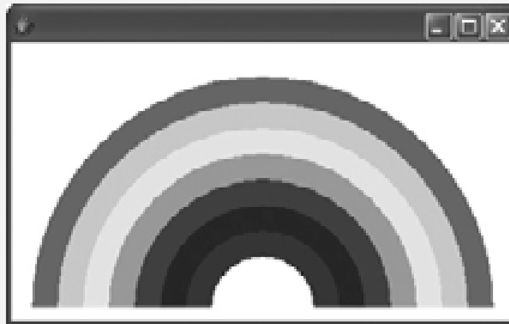


Figura 7.23 | Creación de un objeto `JFrame` para mostrar un arco iris.

grados, en donde los cero grados apuntan a la derecha. Un barrido positivo dibuja el arco en sentido contrario a las manecillas del reloj, mientras que un barrido negativo dibuja el arco en sentido de las manecillas del reloj. Un método similar a `fillArc` es `drawArc`; requiere los mismos parámetros que `fillArc`, pero dibuja el borde del arco, en vez de rellenarlo.

La clase `PruebaDibujoArcoIris` (figura 7.23) crea y establece un objeto `JFrame` para mostrar el arco iris en la pantalla. Una vez que el programa hace visible el objeto `JFrame`, el sistema llama al método `paintComponent` en la clase `DibujoArcoIris` para dibujar el arco iris en la pantalla.

Ejercicio del ejemplo práctico de GUI y gráficos

7.1 (*Dibujo de espirales*) En este ejercicio, dibujará espirales con los métodos `drawLine` y `drawArc`.

a) Dibuje una espiral con forma cuadrada (como en la captura de pantalla izquierda de la figura 7.24), centrada en el panel, usando el método `drawLine`. Una técnica es utilizar un ciclo que incremente la longitud de la línea después de dibujar cada segunda línea. La dirección en la cual se dibujará la siguiente línea debe ir después de un patrón distinto, como abajo, izquierda, arriba, derecha.

b) Dibuje una espiral circular (como en la captura de pantalla derecha de la figura 7.24), usando el método `drawArc` para dibujar un semicírculo a la vez. Cada semicírculo sucesivo deberá tener un radio más grande (según lo especificado mediante la anchura del rectángulo delimitador) y debe seguir dibujando en donde terminó el semicírculo anterior.

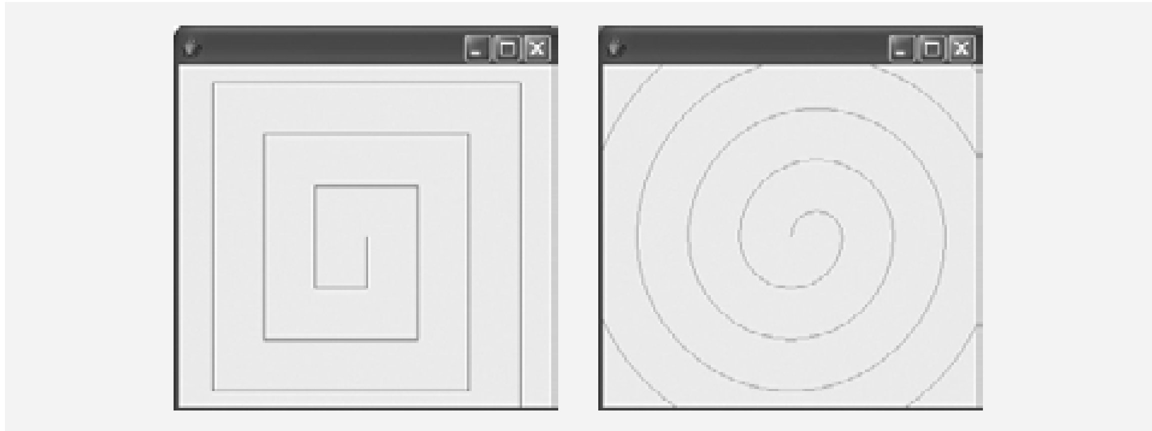


Figura 7.24 | Dibujo de una espiral usando `drawLine` (izquierda) y `drawArc` (derecha).

7.14 (Opcional) Ejemplo práctico de Ingeniería de Software: colaboración entre los objetos

En esta sección nos concentraremos en las colaboraciones (interacciones) entre los objetos. Cuando dos objetos se comunican entre sí para realizar una tarea, se dice que **colaboran** (para ello, un objeto invoca a las operaciones del otro). Una **colaboración** consiste en que un objeto de una clase envía un **mensaje** a un objeto de otra clase. En Java, los mensajes se envían mediante llamadas a métodos.

En la sección 6.14 determinamos muchas de las operaciones de las clases en nuestro sistema. En esta sección, nos concentraremos en los mensajes que invocan a esas operaciones. Para identificar las colaboraciones en el sistema, regresaremos al documento de requerimientos de la sección 2.9. Recuerde que este documento especifica el rango de actividades que ocurren durante una sesión con el ATM (por ejemplo, autenticar a un usuario, realizar transacciones). Los pasos utilizados para describir cómo debe realizar el sistema cada una de estas tareas son nuestra primera indicación de las colaboraciones en nuestro sistema. A medida que avancemos por ésta y las siguientes secciones del Ejemplo práctico de Ingeniería de Software que quedan en el libro, es probable que descubramos relaciones adicionales.

Identificar las colaboraciones en un sistema

Para identificar las colaboraciones en el sistema, leeremos con cuidado las secciones del documento de requerimientos que especifican lo que debe hacer el ATM para autenticar un usuario, y para realizar cada tipo de transacción. Para cada acción o paso descrito en el documento de requerimientos, decidimos qué objetos en nuestro sistema deben interactuar para lograr el resultado deseado. Identificamos un objeto como el emisor y otro como el receptor. Después seleccionamos una de las operaciones del objeto receptor (identificadas en la sección 6.14) que el objeto emisor debe invocar para producir el comportamiento apropiado. Por ejemplo, el ATM muestra un mensaje de bienvenida cuando está inactivo. Sabemos que un objeto de la clase `Pantalla` muestra un mensaje al usuario a través de su operación `mostrarMensaje`. Por ende, decidimos que el sistema puede mostrar un mensaje de bienvenida si empleamos una colaboración entre el `ATM` y la `Pantalla`, en donde el `ATM` envía un mensaje `mostrarMensaje` a la `Pantalla` mediante la invocación de la operación `mostrarMensaje` de la clase `Pantalla`. [*Nota:* para evitar repetir la frase "un objeto de la clase…", nos referiremos a cada objeto sólo utilizando su nombre de clase, precedido por un artículo (por ejemplo, "un", "una", "el" o "la"); por ejemplo, "el `ATM`" hace referencia a un objeto de la clase `ATM`].

La figura 7.25 lista las colaboraciones que pueden derivarse del documento de requerimientos. Para cada objeto emisor, listamos las colaboraciones en el orden en el que ocurren primero durante una sesión con el ATM (es decir, el orden en el que se describen en el documento de requerimientos). Listamos cada colaboración en la que se involucre un emisor único, un mensaje y un receptor sólo una vez, aun cuando la colaboración puede ocurrir varias veces durante una sesión con el ATM. Por ejemplo, la primera fila en la figura 7.25 indica que el objeto `ATM` colabora con el objeto `Pantalla` cada vez que el `ATM` necesita mostrar un mensaje al usuario.

Un objeto de la clase...	envía el mensaje...	a un objeto de la clase...
`ATM`	`mostrarMensaje`	`Pantalla`
	`obtenerEntrada`	`Teclado`
	`autenticarUsuario`	`BaseDatosBanco`
	`ejecutar`	`SolicitudSaldo`
	`ejecutar`	`Retiro`
	`ejecutar`	`Deposito`
`SolicitudSaldo`	`obtenerSaldoDisponible`	`BaseDatosBanco`
	`obtenerSaldoTotal`	`BaseDatosBanco`
	`mostrarMensaje`	`Pantalla`
`Retiro`	`MostrarMensaje`	`Pantalla`
	`obtenerEntrada`	`Teclado`
	`obtenerSaldoDisponible`	`BaseDatosBanco`
	`haySuficienteEfectivoDisponible`	`DispensadorEfectivo`
	`cargar`	`BaseDatosBanco`
	`dispensarEfectivo`	`DispensadorEfectivo`
`Deposito`	`mostrarMensaje`	`Pantalla`
	`obtenerEntrada`	`Teclado`
	`seRecibioSobreDeposito`	`RanuraDeposito`
	`abonar`	`BaseDatosBanco`
`BaseDatosBanco`	`validarNIP`	`Cuenta`
	`obtenerSaldoDisponible`	`Cuenta`
	`obtenerSaldoTotal`	`Cuenta`
	`cargar`	`Cuenta`
	`abonar`	`Cuenta`

Figura 7.25 | Colaboraciones en el sistema ATM.

Consideraremos las colaboraciones en la figura 7.25. Antes de permitir que un usuario realice transacciones, el `ATM` debe pedirle que introduzca un número de cuenta y que después introduzca un NIP. Para realizar cada una de estas tareas envía un mensaje a la `Pantalla` a través de `mostrarMensaje`. Ambas acciones se refieren a la misma colaboración entre el `ATM` y la `Pantalla`, que ya se listan en la figura 7.25. El `ATM` obtiene la entrada en respuesta a un indicador, mediante el envío de un mensaje `obtenerEntrada` del `Teclado`. A continuación, el `ATM` debe determinar si el número de cuenta especificado por el usuario y el NIP coinciden con los de una cuenta en la base de datos. Para ello envía un mensaje `autenticarUsuario` a la `BaseDatosBanco`. Recuerde que `BaseDatosBanco` no puede autenticar a un usuario en forma directa; sólo la `Cuenta` del usuario (es decir, la `Cuenta` que contiene el número de cuenta especificado por el usuario) puede acceder al NIP registrado del usuario para autenticarlo. Por lo tanto, la figura 7.25 lista una colaboración en la que `BaseDatosBanco` envía un mensaje `validarNIP` a una `Cuenta`.

Una vez autenticado el usuario, el `ATM` muestra el menú principal enviando una serie de mensajes `mostrarMensaje` a la `Pantalla` y obtiene la entrada que contiene una selección de menú; para ello envía un mensaje `obtenerEntrada` al `Teclado`. Ya hemos tomado en cuenta estas colaboraciones, por lo que no agregamos nada a la figura 7.25. Una vez que el usuario selecciona un tipo de transacción a realizar, el `ATM` ejecuta la transacción enviando un mensaje `ejecutar` a un objeto de la clase de transacción apropiada (es decir, un objeto `SolicitudSaldo`, `Retiro` o `Deposito`). Por ejemplo, si el usuario elije realizar una solicitud de saldo, el `ATM` envía un mensaje `ejecutar` a un objeto `SolicitudSaldo`.

Un análisis más a fondo del documento de requerimientos revela las colaboraciones involucradas en la ejecución de cada tipo de transacción. Un objeto `SolicitudSaldo` extrae la cantidad de dinero disponible en la cuenta del usuario, al enviar un mensaje `obtenerSaldoDisponible` al objeto `BaseDatosBanco`, el cual responde enviando un mensaje `obtenerSaldoDisponible` a la `Cuenta` del usuario. De manera similar, el objeto `SolicitudSaldo` extrae la cantidad de dinero depositado al enviar un mensaje `obtenerSaldoTotal` al

objeto BaseDatosBanco, el cual envía el mismo mensaje a la Cuenta del usuario. Para mostrar en pantalla ambas cantidades del saldo del usuario al mismo tiempo, el objeto SolicitudSaldo envía a la Pantalla un mensaje mostrarMensaje.

Un objeto Retiro envía a la Pantalla una serie de mensajes mostrarMensaje para mostrar un menú de montos estándar de retiro (es decir, $20, $40, $60, $100, $200). El objeto Retiro envía al Teclado un mensaje obtenerEntrada para obtener la selección del menú elegida por el usuario. A continuación, el objeto Retiro determina si el monto de retiro solicitado es menor o igual al saldo de la cuenta del usuario. Para obtener el monto de dinero disponible en la cuenta del usuario, el objeto Retiro envía un mensaje obtenerSaldoDisponible al objeto BaseDatosBanco. Después el objeto Retiro evalúa si el dispensador contiene suficiente efectivo, enviando al DispensadorEfectivo un mensaje haySuficienteEfectivoDisponible. Un objeto Retiro envía un mensaje cargar al objeto BaseDatosBanco para reducir el saldo de la cuenta del usuario. El objeto BaseDatosBanco envía a su vez el mismo mensaje al objeto Cuenta apropiado. Recuerde que al hacer un cargo a una Cuenta se reduce tanto el saldo total como el saldo disponible. Para dispensar la cantidad solicitada de efectivo, el objeto Retiro envía un mensaje dispensarEfectivo al objeto DispensadorEfectivo. Por último, el objeto Retiro envía a la Pantalla un mensaje mostrarMensaje, instruyendo al usuario para que tome el efectivo.

Para responder a un mensaje ejecutar, un objeto Deposito primero envía a la Pantalla un mensaje mostrarMensaje para pedir al usuario que introduzca un monto a depositar. El objeto Deposito envía al Teclado un mensaje obtenerEntrada para obtener la entrada del usuario. Después, el objeto Deposito envía a la Pantalla un mensaje mostrarMensaje para pedir al usuario que inserte un sobre de depósito. Para determinar si la ranura de depósito recibió un sobre de depósito entrante, el objeto Deposito envía al objeto RanuraDeposito un mensaje seRecibioSobreDeposito. El objeto Deposito actualiza la cuenta del usuario enviando un mensaje abonar al objeto BaseDatosBanco, el cual a su vez envía un mensaje abonar al objeto Cuenta del usuario. Recuerde que al abonar a una Cuenta se incrementa el saldoTotal, pero no el saldoDisponible.

Diagramas de interacción

Ahora que identificamos un conjunto de posibles colaboraciones entre los objetos en nuestro sistema ATM, modelaremos en forma gráfica estas interacciones. UML cuenta con varios tipos de **diagramas de interacción**, que para modelar el comportamiento de un sistema modelan la forma en que los objetos interactúan entre sí. El **diagrama de comunicación** enfatiza *cuáles objetos* participan en las colaboraciones. [*Nota:* en versiones anteriores de UML los diagramas de comunicación se llamaban **diagramas de colaboración**]. Al igual que el diagrama de comunicación, el **diagrama de secuencia** muestra las colaboraciones entre los objetos, pero enfatiza *cuándo* se deben enviar los mensajes entre los objetos *a través del tiempo*.

Diagramas de comunicación

La figura 7.26 muestra un diagrama de comunicación que modela la forma en que el ATM ejecuta una SolicitudSaldo. Los objetos se modelan en UML como rectángulos que contienen nombres de la forma nombreObjeto : NombreClase. En este ejemplo, que involucra sólo a un objeto de cada tipo, descartamos el nombre del objeto y listamos sólo un signo de dos puntos (:) seguido del nombre de la clase. [*Nota:* se recomienda especificar el nombre de cada objeto en un diagrama de comunicación cuando se modelan varios objetos del mismo tipo]. Los objetos que se comunican se conectan con líneas sólidas y los mensajes se pasan entre los objetos a lo largo de estas líneas, en la dirección mostrada por las flechas. El nombre del mensaje, que aparece enseguida de la flecha, es el nombre de una operación (es decir, un método en Java) que pertenece al objeto receptor; considere el nombre como un "servicio" que el objeto receptor proporciona a los objetos emisores (sus "clientes").

La flecha rellena en la figura 7.26 representa un mensaje (o **llamada síncrona**) en UML y una llamada a un método en Java. Esta flecha indica que el flujo de control va desde el objeto emisor (el ATM) hasta el objeto receptor (una SolicitudSaldo). Como ésta es una llamada síncrona, el objeto emisor no puede enviar otro mensaje, ni hacer cualquier otra cosa, hasta que el objeto receptor procese el mensaje y devuelva el control al objeto

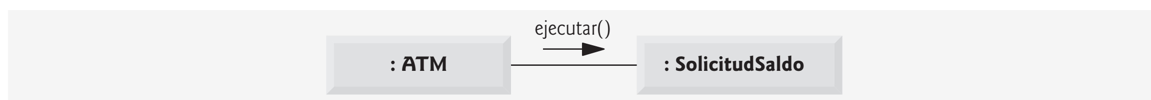


Figura 7.26 | Diagrama de comunicación del ATM, ejecutando una solicitud de saldo.

emisor. El emisor sólo espera. Por ejemplo, en la figura 7.26 el objeto `ATM` llama al método `ejecutar` de un objeto `SolicitudSaldo` y no puede enviar otro mensaje sino hasta que `ejecutar` termine y devuelva el control al objeto `ATM`. [*Nota:* si ésta fuera una **llamada asíncrona**, representada por una flecha, el objeto emisor no tendría que esperar a que el objeto receptor devolviera el control; continuaría enviando mensajes adicionales inmediatamente después de la llamada asíncrona. Dichas llamadas se implementan en Java mediante el uso de una técnica conocida como subprocesamiento múltiple, que veremos en el capítulo 23, Subprocesamiento múltiple].

Secuencia de mensajes en un diagrama de comunicación

La figura 7.27 muestra un diagrama de comunicación que modela las interacciones entre los objetos en el sistema, cuando se ejecuta un objeto de la clase `SolicitudSaldo`. Asumimos que el atributo `numeroCuenta` del objeto contiene el número de cuenta del usuario actual. Las colaboraciones en la figura 7.27 empiezan después de que el objeto `ATM` envía un mensaje `ejecutar` a un objeto `SolicitudSaldo` (es decir, la interacción modelada en la figura 7.26). El número a la izquierda del nombre de un mensaje indica el orden en el que éste se pasa. La **secuencia de mensajes** en un diagrama de comunicación progresa en orden numérico, de menor a mayor. En este diagrama, la numeración comienza con el mensaje `1` y termina con el mensaje `3`. El objeto `SolicitudSaldo` envía primero un mensaje `obtenerSaldoDisponible` al objeto `BaseDatosBanco` (mensaje `1`), después envía un mensaje `obtenerSaldoTotal` al objeto `BaseDatosBanco` (mensaje `2`). Dentro de los paréntesis que van después del nombre de un mensaje, podemos especificar una lista separada por comas de los nombres de los parámetros que se envían con el mensaje (es decir, los argumentos en una llamada a un método en Java); el objeto `SolicitudSaldo` pasa el atributo `numeroCuenta` con sus mensajes al objeto `BaseDatosBanco` para indicar de cuál objeto `Cuenta` se extraerá la información del saldo. En la figura 6.22 vimos que las operaciones `obtenerSaldoDisponible` y `obtenerSaldoTotal` de la clase `BaseDatosBanco` requieren cada una de ellas un parámetro para identificar una cuenta. El objeto `SolicitudSaldo` muestra a continuación el `saldoDisponible` y el `saldoTotal` al usuario; para ello pasa un mensaje `mostrarMensaje` a la `Pantalla` (mensaje `3`) que incluye un parámetro, el cual indica el `mensaje` a mostrar.

Observe que la figura 7.27 modela dos mensajes adicionales que se pasan del objeto `BaseDatosBanco` a un objeto `Cuenta` (mensaje `1.1` y mensaje `2.1`). para proveer al `ATM` los dos saldos de la `Cuenta` del usuario (según lo solicitado por los mensajes `1` y `2`), el objeto `BaseDatosBanco` debe pasar un mensaje `obtenerSaldoDisponible` y un mensaje `obtenerSaldoTotal` a la `Cuenta` del usuario. Dichos mensajes que se pasan dentro del manejo de otro mensaje se llaman **mensajes anidados**. UML recomienda utilizar un esquema de numeración decimal para indicar mensajes anidados. Por ejemplo, el mensaje `1.1` es el primer mensaje anidado en el mensaje `1`; el objeto `BaseDatosBanco` pasa un mensaje `obtenerSaldoDisponible` durante el procesamiento de `BaseDatosBanco`

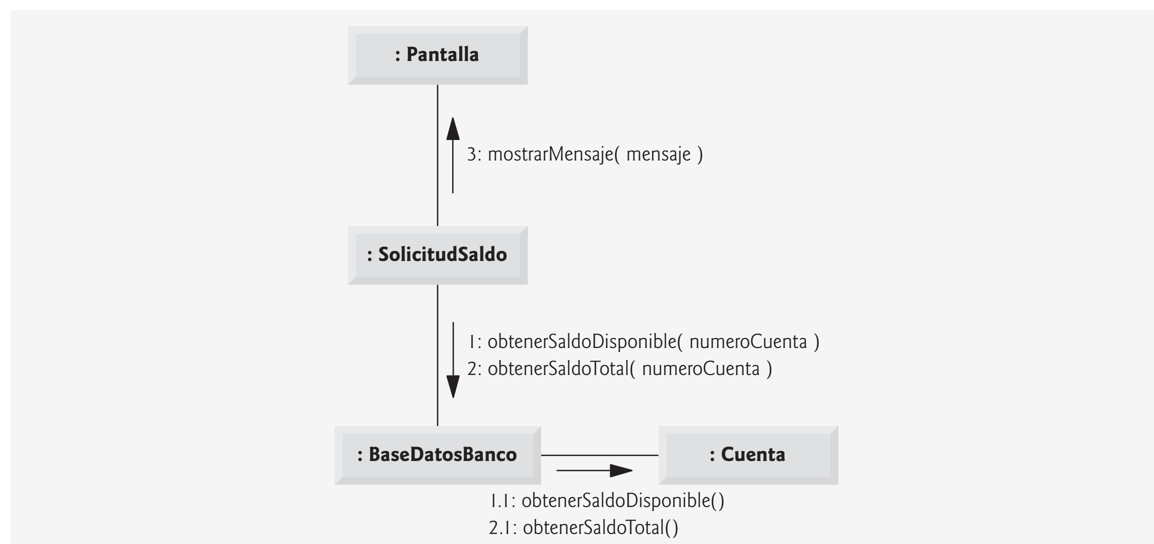


Figura 7.27 | Diagrama de comunicación para ejecutar una solicitud de saldo.

de un mensaje con el mismo nombre. [*Nota:* si el objeto `BaseDatosBanco` necesita pasar un segundo mensaje anidado mientras procesa el mensaje `1`, el segundo mensaje se numera como `1.2`]. Un mensaje puede pasarse sólo cuando se han pasado ya todos los mensajes anidados del mensaje anterior. Por ejemplo, el objeto `Solicitud-Saldo` pasa el mensaje `3` sólo hasta que se han pasado los mensajes `2` y `2.1`, en ese orden.

El esquema de numeración anidado que se utiliza en los diagramas de comunicación ayuda a aclarar con precisión cuándo y en qué contexto se pasa cada mensaje. Por ejemplo, si numeramos los cinco mensajes de la figura 7.27 usando un esquema de numeración plano (es decir, `1`, `2`, `3`, `4`, `5`), podría ser posible que alguien que viera el diagrama no pudiera determinar que el objeto `BaseDatosBanco` pasa el mensaje `obtenerSaldoDisponible` (mensaje `1.1` a una `Cuenta` *durante* el procesamiento del mensaje `1` por parte del objeto `BaseDatosBanco`, en vez de hacerlo *después* de completar el procesamiento del mensaje `1`. Los números decimales anidados hacen ver que el segundo mensaje `obtenerSaldoDisponible` (mensaje `1.1`) se pasa a una `Cuenta` dentro del manejo del primer mensaje `obtenerSaldoDisponible` (mensaje `1`) por parte del objeto `BaseDatosBanco`.

Diagramas de secuencia

Los diagramas de comunicación enfatizan los participantes en las colaboraciones, pero modelan su sincronización de una forma bastante extraña. Un diagrama de secuencia ayuda a modelar la sincronización de las colaboraciones con más claridad. La figura 7.28 muestra un diagrama de secuencia que modela la secuencia de las interacciones que ocurren cuando se ejecuta un `Retiro`. La línea punteada que se extiende hacia abajo desde el rectángulo de un objeto es la **línea de vida** de ese objeto, la cual representa la evolución en el tiempo. Las acciones ocurren a lo largo de la línea de vida de un objeto, en orden cronológico de arriba hacia abajo; una acción cerca de la parte superior ocurre antes que una cerca de la parte inferior.

El paso de mensajes en los diagramas de secuencia es similar al paso de mensajes en los diagramas de comunicación. Una flecha con punta rellena, que se extiende desde el objeto emisor hasta el objeto receptor, representa un mensaje entre dos objetos. La punta de flecha apunta a una activación en la línea de vida del objeto receptor. Una **activación**, que se muestra como un rectángulo vertical delgado, indica que se está ejecutando un objeto. Cuando un objeto devuelve el control, un mensaje de retorno (representado como una línea punteada con una punta de flecha) se extiende desde la activación del objeto que devuelve el control hasta la activación del objeto que envió originalmente el mensaje. Para eliminar el desorden, omitimos las flechas de los mensajes de retorno; UML permite esta práctica para que los diagramas sean más legibles. Al igual que los diagramas de comunicación, los de secuencia pueden indicar parámetros de mensaje entre los paréntesis que van después del nombre de un mensaje.

La secuencia de mensajes de la figura 7.28 empieza cuando un objeto `Retiro` pide al usuario que seleccione un monto de retiro; para ello envía a la `Pantalla` un mensaje `mostrarMensaje`. Después el objeto `Retiro` envía al `Teclado` un mensaje `obtenerEntrada`, el cual obtiene los datos de entrada del usuario. En el diagrama de actividad de la figura 5.31 modelamos la lógica de control involucrada en un objeto `Retiro`, por lo que no mostraremos esta lógica en el diagrama de secuencia de la figura 7.28. En vez de ello modelaremos el escenario para el mejor caso, en el cual el saldo de la cuenta del usuario es mayor o igual al monto de retiro seleccionado, y el dispensador de efectivo contiene un monto de efectivo suficiente como para satisfacer la solicitud. Para obtener información acerca de cómo modelar la lógica de control en un diagrama de secuencia, consulte los recursos Web y las lecturas recomendadas que se listan al final de la sección 2.9.

Después de obtener un monto de retiro, el objeto `Retiro` envía un mensaje `obtenerSaldoDisponible` al objeto `BaseDatosBanco`, el cual a su vez envía un mensaje `obtenerSaldoDisponible` a la `Cuenta` del usuario. Suponiendo que la cuenta del usuario tiene suficiente dinero disponible para permitir la transacción, el objeto `Retiro` envía al objeto `DispensadorEfectivo` un mensaje `haySuficienteEfectivoDisponible`. Suponiendo que hay suficiente efectivo disponible, el objeto `Retiro` reduce el saldo de la cuenta del usuario (tanto el `saldo-Total` como el `saldoDisponible`) enviando un mensaje `cargar` a la `Cuenta` del usuario. Por último, el objeto `Retiro` envía al `DispensadorEfectivo` un mensaje `dispensarEfectivo` y a la `Pantalla` un mensaje `mostrar-Mensaje`, indicando al usuario que retire el efectivo de la máquina.

Hemos identificado las colaboraciones entre los objetos en el sistema ATM, y modelamos algunas de estas colaboraciones usando los diagramas de interacción de UML: los diagramas de comunicación y los diagramas de secuencia. En la siguiente sección del Ejemplo práctico de Ingeniería de Software (sección 8.19), mejoraremos la estructura de nuestro modelo para completar un diseño orientado a objetos preliminar, y después empezaremos a implementar el sistema ATM en Java.

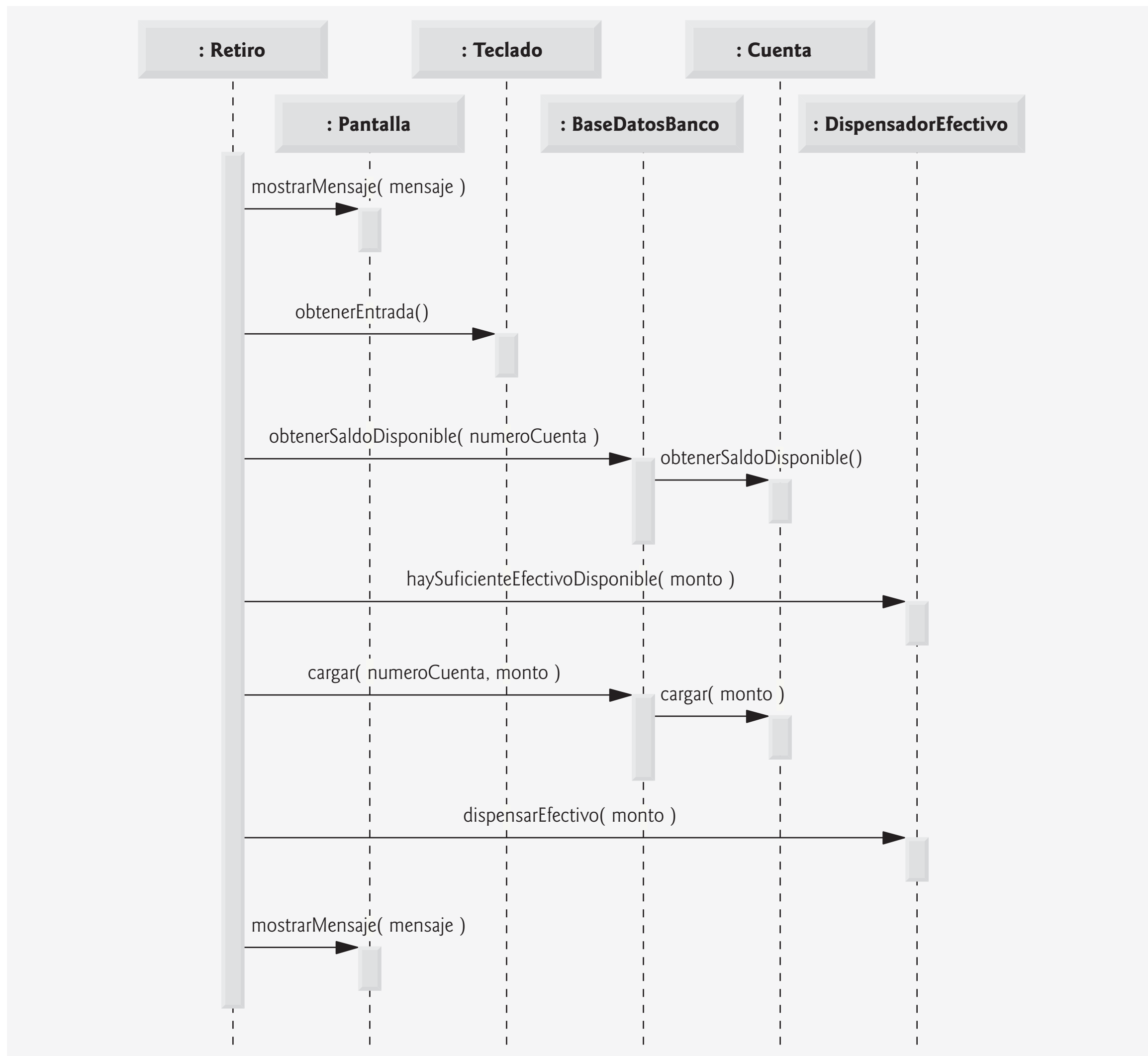


Figura 7.28 | Diagrama de secuencia que modela la ejecución de un `Retiro`.

Ejercicios de autoevaluación del Ejemplo práctico de Ingeniería de Software

7.1 Un(a) __________ consiste en que un objeto de una clase envía un mensaje a un objeto de otra clase.
a) asociación
b) agregación
c) colaboración
d) composición

7.2 ¿Cuál forma de diagrama de interacción es la que enfatiza *qué* colaboraciones se llevan a cabo? ¿Cuál forma enfatiza *cuándo* ocurren las interacciones?

7.3 Cree un diagrama de secuencia para modelar las interacciones entre los objetos del sistema ATM, que ocurran cuando se ejecute un `Deposito` con éxito. Explique la secuencia de los mensajes modelados por el diagrama.

Respuestas a los ejercicios de autoevaluación del Ejemplo práctico de Ingeniería de Software

7.1 c.

7.2 Los diagramas de comunicación enfatizan *qué* colaboraciones se llevan a cabo. Los diagramas de secuencia enfatizan *cuándo* ocurren las colaboraciones.

7.3 La figura 7.29 presenta un diagrama de secuencia que modela las interacciones entre objetos en el sistema ATM, las cuales ocurren cuando un `Deposito` se ejecuta con éxito. La figura 7.29 indica que un `Deposito` primero envía un mensaje `mostrarMensaje` a la `Pantalla`, para pedir al usuario que introduzca un monto de depósito. A continuación, el `Deposito` envía un mensaje `obtenerEntrada` al `Teclado` para recibir la entrada del usuario. Después, el `Deposito` pide al usuario que inserte un sobre de depósito; para ello envía un mensaje `mostrarMensaje` a la `Pantalla`. Luego, el `Deposito` envía un mensaje `seRecibioSobreDeposito` al objeto `RanuraDeposito` para confirmar que el ATM haya recibido el sobre de depósito. Por último, el objeto `Deposito` incrementa el atributo `saldoTotal` (pero no el atributo `saldoDisponible`) de la `Cuenta` del usuario, enviando al objeto `BaseDatosBanco` un mensaje `abonar`. El objeto `BaseDatosBanco` responde enviando el mismo mensaje a la `Cuenta` del usuario.

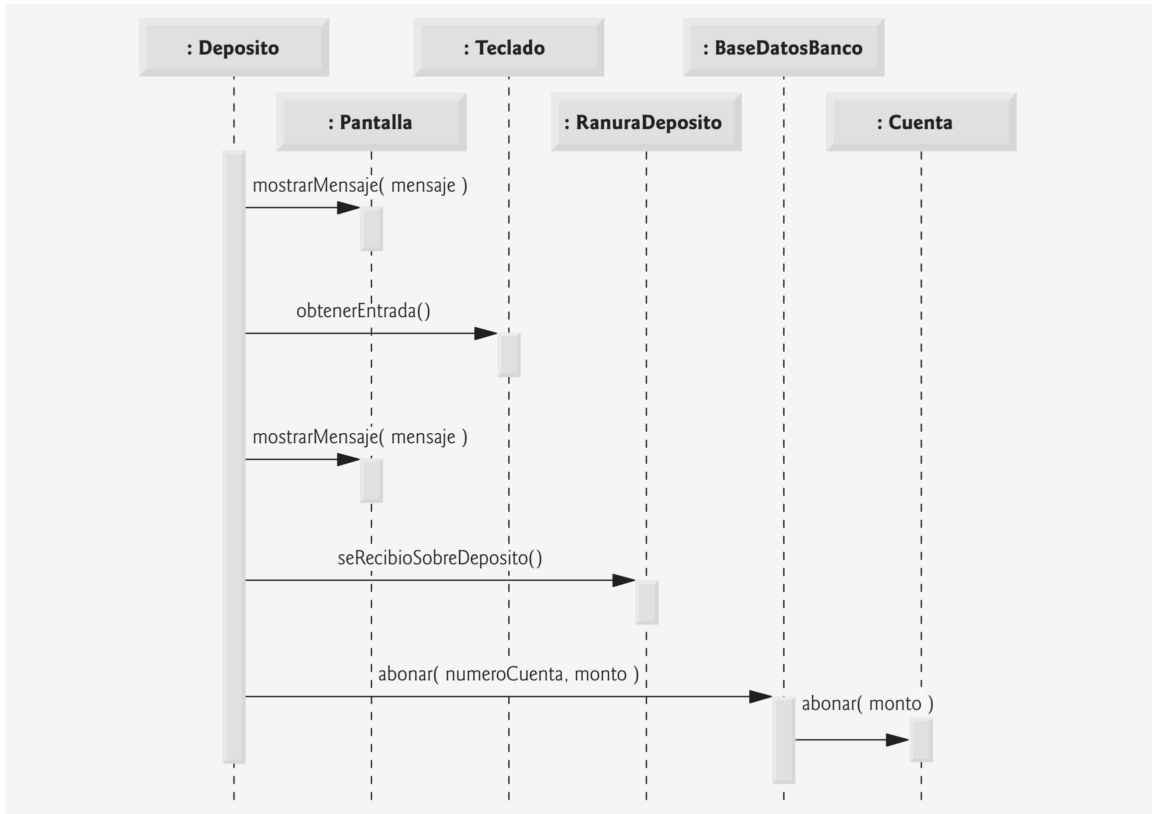


Figura 7.29 | Diagrama de secuencia que modela la ejecución de un `Deposito`.

7.15 Conclusión

En este capítulo empezó nuestra introducción a las estructuras de datos, explorando el uso de los arreglos para almacenar datos y obtenerlos de listas y tablas de valores. Los ejemplos de este capítulo demostraron cómo declarar un arreglo, inicializarlo y hacer referencia a los elementos individuales de un arreglo. Se introdujo la instrucción `for` mejorada para iterar a través de los arreglos. También le mostramos cómo pasar arreglos a los métodos, y cómo declarar y manipular arreglos multidimensionales. Por último, en este capítulo se demostró cómo escribir métodos que utilizan listas de argumentos de longitud variable, y cómo leer argumentos que se pasan a un programa desde la línea de comandos.

Continuaremos con nuestra cobertura de las estructuras de datos en el capítulo 17, Estructuras de datos, que introduce las estructuras de datos dinámicas, como listas, colas, pilas y árboles, que pueden crecer y reducirse a medida que se ejecutan los programas. En el capítulo 18, Genéricos, se presenta el tema de los genéricos, que proporcionan los medios para crear modelos generales de métodos y clases que pueden declararse una vez, pero

utilizarse con muchos tipos de datos distintos. En el capítulo 19, Colecciones, se introduce el Java Collections Framework (Marco de trabajo de colecciones de Java), que utiliza los genéricos para permitir a los programadores especificar los tipos exactos de objetos que almacenará una estructura de datos específica. En el capítulo 19 también se introducen las estructuras de datos predefinidas de Java, que los programadores pueden usar en vez de tener que construir sus propias estructuras. El capítulo 19 habla sobre muchas clases de estructuras de datos, incluyendo `Vector` y `ArrayList`, que son estructuras de datos similares a los arreglos y pueden aumentar y reducir su tamaño en respuesta al cambio en los requerimientos de almacenamiento de un programa. La API Collections también proporciona la clase `Arrays`, que contiene métodos utilitarios para la manipulación de arreglos. El capítulo 19 utiliza varios métodos `static` de la clase `Arrays` para realizar manipulaciones como ordenar y buscar en los datos de un arreglo. Después de leer este capítulo, usted podrá utilizar algunos de los métodos de `Arrays` que se describen en el capítulo 19, pero algunos de los métodos de `Arrays` requieren un conocimiento sobre los conceptos que presentaremos más adelante en este libro.

Ya le hemos presentado los conceptos básicos de las clases, los objetos, las instrucciones de control, los métodos y los arreglos. En el capítulo 8 analizaremos con más detalle las clases y los objetos.

Resumen

Sección 7.1 Introducción

- Los arreglos son estructuras de datos que consisten en elementos de datos relacionados del mismo tipo; son entidades de longitud fija; permanecen con la misma longitud una vez que se crean, aunque a una variable tipo arreglo se le puede reasignar la referencia de un nuevo arreglo de una longitud distinta.

Sección 7.2 Arreglos

- Un arreglo es un grupo de variables (llamadas elementos o componentes) que contienen valores, todos con el mismo tipo. Los arreglos son objetos, por lo cual se consideran como tipos por referencia. Los elementos de un arreglo pueden ser tipos primitivos o tipos por referencia (incluyendo arreglos).
- Para hacer referencia a un elemento específico en un arreglo, especificamos el nombre de la referencia al arreglo y el índice (subíndice) del elemento en el arreglo.
- Un programa hace referencia a cualquiera de los elementos de un arreglo mediante una expresión de acceso a un arreglo, la cual incluye el nombre del arreglo, seguido del subíndice del elemento específico entre corchetes (`[]`).
- El primer elemento en cada arreglo tiene el subíndice cero, y algunas veces se le llama el elemento cero.
- Un subíndice debe ser un entero positivo. Un programa puede utilizar una expresión como un subíndice.
- Todo objeto tipo arreglo conoce su propia longitud, y mantiene esta información en un campo `length`. La expresión *arreglo*`.length` accede al campo `length` de *arreglo*, para determinar la longitud del arreglo.

Sección 7.3 Declaración y creación de arreglos

- Para crear un objeto tipo arreglo, el programador especifica el tipo de los elementos del arreglo y el número de elementos como parte de una expresión de creación de arreglo, que utiliza la palabra clave `new`. La siguiente expresión de creación de arreglo crea un arreglo de 100 valores `int`:

```
int b[ ] = new int[ 100 ];
```

- Cuando se crea un arreglo, cada elemento del mismo recibe un valor predeterminado: cero para los elementos numéricos de tipo primitivo, `false` para los elementos booleanos y `null` para las referencias (cualquier tipo no primitivo).
- Cuando se declara un arreglo, su tipo y los corchetes pueden combinarse al principio de la declaración, para indicar que todos los identificadores en la declaración son variables tipo arreglo, como en

```
double[ ] arreglo1, arreglo2;
```

- Un programa puede declarar arreglos de cualquier tipo. Cada elemento de un arreglo de tipo primitivo contiene una variable del tipo declarado del arreglo. De manera similar, en un arreglo de un tipo por referencia, cada elemento es una referencia a un objeto del tipo declarado del arreglo.

Sección 7.4 Ejemplos acerca del uso de los arreglos

- Un programa puede crear un arreglo e inicializar sus elementos con un inicializador de arreglos (es decir, una lista inicializadora encerrada entre llaves).
- Las variables constantes (también conocidas como constantes con nombre o variables de sólo lectura) deben inicializarse antes de utilizarlas, y no pueden modificarse de ahí en adelante.
- Cuando se ejecuta un programa en Java, la JVM comprueba los subíndices de los arreglos para asegurarse que sean válidos (es decir, deben ser mayores o iguales a `0` y menores que la longitud del arreglo). Si un programa utiliza un subíndice inválido, Java genera algo que se conoce como excepción, para indicar que ocurrió un error en el programa, en tiempo de ejecución.

Sección 7.6 Instrucción `for` mejorada

- La instrucción `for` mejorada permite a los programadores iterar a través de los elementos de un arreglo o de una colección, sin utilizar un contador. La sintaxis de una instrucción `for` mejorada es:

  `for` (*parámetro* : *nombreArreglo*)
  instrucción

  en donde *parámetro* tiene dos partes: un tipo y un identificador (por ejemplo, `int numero`), y *nombreArreglo* es el arreglo a través del cual se iterará.
- La instrucción `for` mejorada no puede usarse para modificar los elementos de un arreglo. Si un programa necesita modificar elementos, use la instrucción `for` tradicional, controlada por contador.

Sección 7.7 Paso de arreglos a los métodos

- Cuando un argumento se pasa por valor, se hace una copia del valor del argumento y se pasa al método que se llamó. Este método trabaja exclusivamente con la copia.
- Cuando se pasa un argumento por referencia, el método al que se llamó puede acceder al valor del argumento en el método que lo llamó directamente, y es posible que pueda modificarlo.
- Java no permite a los programadores elegir entre el paso por valor y el paso por referencia; todos los argumentos se pasan por valor. Una llamada a un método puede pasar dos tipos de valores a un método: copias de valores primitivos (por ejemplo, valores de tipo `int` y `double`) y copias de referencias a objetos. Aunque la referencia a un objeto se pasa por valor, un método de todas formas puede interactuar con el objeto referenciado, llamando a sus métodos `public` mediante el uso de la copia de la referencia al objeto.
- Para pasar a un método una referencia a un objeto, simplemente se especifica en la llamada al método el nombre de la variable que hace referencia al objeto.
- Cuando un argumento para un método es todo un arreglo, o un elemento individual del arreglo de tipo por referencia, el método que se llamó recibe una copia del arreglo o referencia al elemento. Cuando un argumento para un método es un elemento individual de un arreglo de tipo primitivo, el método que se llamó recibe una copia del valor del elemento.
- Para pasar un elemento individual del arreglo a un método, use el nombre indexado del arreglo como argumento en la llamada al método.

Sección 7.9 Arreglos multidimensionales

- Los arreglos multidimensionales con dos dimensiones se utilizan a menudo para representar tablas de valores, que consisten en información ordenada en filas y columnas.
- Los arreglos que requieren dos subíndices para identificar un elemento específico se llaman arreglos bidimensionales. Un arreglo con *m* filas y *n* columnas se llama arreglo de *m* por *n*. Un arreglo bidimensional puede inicializarse con un inicializador de arreglos, de la forma

  tipoArreglo nombreArreglo`[][] = { {` *fila1 inicializador* `}, {` *fila2 inicializador* `}, … };`

- Los arreglos multidimensionales se mantienen como arreglos de arreglos unidimensionales separados. Como resultado, no es obligatorio que las longitudes de las filas en un arreglo bidimensional sean iguales.
- Un arreglo multidimensional con el mismo número de columnas en cada fila se puede crear mediante una expresión de creación de arreglos, de la forma

  tipoArreglo nombreArreglo`[][] = new` *tipoArreglo*`[` *numFilas* `][` *numColumnas* `];`

- Un tipo de argumento seguido por una elipsis (…) en la lista de parámetros de un método indica que éste recibe un número variable de argumentos de ese tipo específico. La elipsis puede ocurrir sólo una vez en la lista de parámetros de un método, y debe estar al final de la lista.

Sección 7.11 Listas de argumentos de longitud variable

- Una lista de argumentos de longitud variable se trata como un arreglo dentro del cuerpo del método. El número de argumentos en el arreglo se puede obtener mediante el campo `length` del arreglo.

Sección 7.12 Uso de argumentos de línea de comandos

- Para pasar argumentos a `main` en una aplicación de Java, desde la línea de comandos, se incluye un parámetro de tipo `String[ ]` en la lista de parámetros de `main`. Por convención, el parámetro de `main` se llama `args`.
- Java pasa los argumentos de línea de comandos que aparecen después del nombre de la clase en el comando `java` al método `main` de la aplicación, en forma de objetos `String` en el arreglo `args`. El número de argumentos que se pasan de la línea de comandos se obtiene accediendo al atributo `length` del arreglo.

Terminología

`0`, bandera (en un especificador de formato)
`a[ i ]`
`a[ i ][ j ]`
argumentos de línea de comandos
arreglo
arreglo bidimensional
arreglo de *m* por *n*
arreglo multidimensional
arreglo unidimensional
cantidad escalar
columna de un arreglo bidimensional
componente de un arreglo
comprobación de límites
constante con nombre
corchetes, `[]`
declarar un arreglo
declarar una variable constante
elemento de un arreglo
elipsis (...) en la lista de parámetros de un método
error de desplazamiento por uno
escalar
estructura de datos
expresión de acceso a un arreglo
expresión de creación de arreglos
fila de un arreglo bidimensional
`final`, palabra clave
formato tabular
índice
índice de columna
inicializador de arreglos
inicializadores de arreglos anidados
inicializar un arreglo
instrucción `for` mejorada
`length`, campo de un arreglo
lista de argumentos de longitud variable
lista inicializadora
llamada por referencia
llamada por valor
nombre de un arreglo
número de posición
`parseInt`, método de la clase `Integer`
paso de arreglos a métodos
paso por referencia
paso por valor
recorrer un arreglo
subíndice
subíndice cero
subíndice de fila
tabla de valores
valor de un elemento
variable constante
variable de sólo lectura

Ejercicios de autoevaluación

7.1 Complete las siguientes oraciones:

a) Las listas y tablas de valores pueden guardarse en ______________.
b) Un arreglo es un grupo de ______________ (llamados elementos o componentes) que contiene valores, todos con el mismo ______________.
c) La ______________ permite a los programadores iterar a través de los elementos en un arreglo, sin utilizar un contador.
d) El número utilizado para referirse a un elemento específico de un arreglo se conoce como el ______________ de ese elemento.
e) Un arreglo que utiliza dos subíndices se conoce como un arreglo ______________.
f) Use la instrucción `for` mejorada ______________ para recorrer el arreglo `double` llamado `numeros`.
g) Los argumentos de línea de comandos se almacenan en ______________.
h) Use la expresión ______________ para recibir el número total de argumentos en una línea de comandos. Suponga que los argumentos de línea de comandos se almacenan en el objeto `String args[]`.

i) Dado el comando `java MiClase prueba`, el primer argumento de línea de comandos es ____________.
j) Un(a) ______________________ en la lista de parámetros de un método indica que el método puede recibir un número variable de argumentos.

7.2 Conteste con *verdadero* o *falso* a cada una de las siguientes proposiciones; en caso de ser *falso*, explique por qué.
a) Un arreglo puede guardar muchos tipos distintos de valores.
b) El índice de un arreglo debe ser generalmente de tipo `float`.
c) Un elemento individual de un arreglo que se pasa a un método y se modifica ahí mismo, contendrá el valor modificado cuando el método llamado termine su ejecución.
d) Los argumentos de línea de comandos se separan por comas.

7.3 Realice las siguientes tareas para un arreglo llamado `fracciones`:
a) Declare una constante llamada `TAMANIO_ARREGLO` que se inicialice con 10.
b) Declare un arreglo con `TAMANIO_ARREGLO` elementos de tipo `double`, e inicialice los elementos con `0`.
c) Haga referencia al elemento 4 del arreglo.
d) Asigne el valor `1.667` al elemento 9 del arreglo.
e) Asigne el valor `3.333` al elemento 6 del arreglo.
f) Sume todos los elementos del arreglo, utilizando una instrucción `for`. Declare la variable entera `x` como variable de control para el ciclo.

7.4 Realice las siguientes tareas para un arreglo llamado `tabla`:
a) Declare y cree el arreglo como un arreglo entero con tres filas y tres columnas. Suponga que se ha declarado la constante `TAMANIO_ARREGLO` con el valor de `3`.
b) ¿Cuántos elementos contiene el arreglo?
c) Utilice una instrucción `for` para inicializar cada elemento del arreglo con la suma de sus índices. Suponga que se declaran las variables enteras `x` y `y` como variables de control.

7.5 Encuentre y corrija el error en cada uno de los siguientes fragmentos de programa:
a)
```
final int TAMANIO_ARREGLO = 5;
TAMANIO_ARREGLO = 10;
```
b) Suponga que `int b[] = new int[ 10 ];`
```
for ( int i = 0; i <= b.length; i++ )
   b[i] = 1;
```
c) Suponga que int `a[ ][ ] = { { 1, 2 }, { 3, 4 } };`
```
a[ 1, 1 ] = 5;
```

Respuestas a los ejercicios de autoevaluación

7.1 a) arreglos. b) variables, tipo. c) instrucción `for` mejorada. d) índice (o subíndice, o número de posición) e) bidimensional. f) `for ( double d : numeros )`. g) un arreglo de objetos `String`, llamado `args` por convención. h) `args.length`. i) `prueba`. j) elipsis (…).

7.2 a) Falso. Un arreglo sólo puede guardar valores del mismo tipo.
b) Falso. El índice de un arreglo debe ser un entero o una expresión entera.
c) Para los elementos individuales de tipo primitivo en un arreglo: falso. Un método al que se llama recibe y manipula una copia del valor de dicho elemento, por lo que las modificaciones no afectan el valor original. No obstante, si se pasa la referencia de un arreglo a un método, las modificaciones a los elementos del arreglo que se hicieron en el método al que se llamó se reflejan sin duda en el original. Para los elementos individuales de un tipo no primitivo: verdadero. Un método al que se llama recibe una copia de la referencia de dicho elemento, y los cambios al objeto referenciado se reflejan en el elemento del arreglo original.
d) Falso. Los argumentos de línea de comandos se separan por espacio en blanco.

7.3 a) `final int TAMANIO_ARREGLO = 10;`
b) `double fracciones[ ] = new double[ TAMANIO_ARREGLO ];`
c) `fracciones[ 4 ]`
d) `fracciones[ 9 ] = 1.667;`
e) `fracciones[ 6 ] = 3.333;`

f) `fracciones[ 6 ] = 3.333;`

g)
```
double total = 0.0;
for ( int x = 0; x < fracciones.length; x++ )
total += fracciones[ x ];
```

7.4 a) `int tabla[][] = new int[ TAMANIO_ARREGLO ][ TAMANIO_ARREGLO ];`

b) Nueve.

c)
```
for ( int x = 0; x < tabla.length; x++ )
   for ( int y = 0; y < tabla[ x ].length; y++ )
      tabla[ x ][ y ] = x + y;
```

7.5 a) Error: asignar un valor a una constante después de inicializarla.
Corrección: asigne el valor correcto a la constante en una declaración `final int TAMANIO_ARREGLO`, o declare otra variable.

b) Error: se está haciendo referencia al elemento de un arreglo que está fuera de los límites del arreglo (`b[10]`).
Corrección: cambie el operador <= por <.

c) Error: la indización del arreglo se está realizando en forma incorrecta.
Corrección: cambie la instrucción por `a[ 1 ][ 1 ] = 5;`.

Ejercicios

7.6 Complete las siguientes oraciones:

a) Un arreglo unidimensional `p` contiene cuatro elementos. Los nombres de esos elementos son ___________, ___________, ___________ y ___________.

b) Al proceso de nombrar un arreglo, declarar su tipo y especificar el número de dimensiones se le conoce como ___________ el arreglo.

c) En un arreglo bidimensional, el primer índice identifica el(la) ___________ de un elemento y el segundo identifica el(la) ___________ de un elemento.

d) Un arreglo de *m* por *n* contiene ___________ filas, ___________ columnas y ___________ elementos.

e) El nombre del elemento en la fila 3 y la columna 5 del arreglo `d` es ___________.

7.7 Conteste con *verdadero* o *falso* a cada una de las siguientes proposiciones; en caso de ser *falso*, explique por qué.

a) Para referirse a una ubicación o elemento específico dentro de un arreglo, especificamos el nombre del arreglo y el valor del elemento específico.

b) La declaración de un arreglo reserva espacio para el mismo.

c) Para indicar que deben reservarse 100 ubicaciones para el arreglo entero `p`, el programador escribe la declaración

```
p[ 100 ];
```

d) Una aplicación que inicializa con cero los elementos de un arreglo con 15 elementos debe contener al menos una instrucción `for`.

e) Una aplicación que sume el total de los elementos de un arreglo bidimensional debe contener instrucciones `for` anidadas.

7.8 Escriba instrucciones en Java que realicen cada una de las siguientes tareas:

a) Mostrar el valor del elemento 6 del arreglo `f`.

b) Inicializar con `8` cada uno de los cinco elementos del arreglo entero unidimensional `g`.

c) Sumar el total de los 100 elementos del arreglo `c` de punto flotante.

d) Copiar el arreglo `a` de 11 elementos en la primera porción del arreglo `b`, el cual contiene 34 elementos.

e) Determinar e imprimir los valores menor y mayor contenidos en el arreglo `w` con 99 elementos de punto flotante.

7.9 Considere un arreglo entero `t` de dos por tres.

a) Escriba una instrucción que declare y cree a `t`.

b) ¿Cuántas filas tiene `t`?

c) ¿Cuántas columnas tiene `t`?
d) ¿Cuántos elementos tiene `t`?
e) Escriba expresiones de acceso para todos los elementos en la fila 1 de `t`.
f) Escriba expresiones de acceso para todos los elementos en la columna 2 de `t`.
g) Escriba una sola instrucción que asigne cero al elemento de `t` en la fila 0 y la columna 1.
h) Escriba una serie de instrucciones que inicialice cada elemento de `t` con cero. No utilice una instrucción de repetición.
i) Escriba una instrucción `for` anidada que inicialice cada elemento de `t` con cero.
j) Escriba una instrucción `for` anidada que reciba como entrada del usuario los valores de los elementos de `t`.
k) Escriba una serie de instrucciones que determine e imprima el valor más pequeño en `t`.
l) Escriba una instrucción `printf` que muestre los elementos de la primera fila de `t`. No utilice repetición.
m) Escriba una instrucción que totalice los elementos de la tercera columna de `t`. No utilice repetición.
n) Escriba una serie de instrucciones para imprimir el contenido de `t` en formato tabular. Enliste los índices de columna como encabezados a lo largo de la parte superior, y enliste los índices de fila a la izquierda de cada fila.

7.10 (*Comisión por ventas*) Utilice un arreglo unidimensional para resolver el siguiente problema: una compañía paga a sus vendedores por comisión. Los vendedores reciben $200 por semana más el 9% de sus ventas totales de esa semana. Por ejemplo, un vendedor que acumule $5000 en ventas en una semana, recibirá $200 más el 9% de $5000, o un total de $650. Escriba una aplicación (utilizando un arreglo de contadores) que determine cuántos vendedores recibieron salarios en cada uno de los siguientes rangos (suponga que el salario de cada vendedor se trunca a una cantidad entera):

a) $200-299
b) $300-399
c) $400-499
d) $500-599
e) $600-699
f) $700-799
g) $800-899
h) $900-999
i) $1000 en adelante

Sintetice los resultados en formato tabular.

7.11 Escriba instrucciones que realicen las siguientes operaciones con arreglos unidimensionales:

a) Asignar cero a los 10 elementos del arreglo `cuentas` de tipo entero.
b) Sumar uno a cada uno de los 15 elementos del arreglo `bono` de tipo entero.
c) Imprimir los cinco valores del arreglo `mejoresPuntuaciones` de tipo entero en formato de columnas.

7.12 (*Eliminación de duplicados*) Use un arreglo unidimensional para resolver el siguiente problema: escriba una aplicación que reciba como entrada cinco números, cada uno de los cuales debe estar entre 10 y 100. A medida que se lea cada número, muéstrelo solamente si no es un duplicado de un número que ya se haya leído. Prepárese para el "peor caso", en el que los cinco números son diferentes. Use el arreglo más pequeño que sea posible para resolver este problema. Muestre el conjunto completo de valores únicos introducidos, después de que el usuario introduzca cada nuevo valor.

7.13 Etiquete los elementos del arreglo bidimensional `ventas` de tres por cinco, para indicar el orden en el que se establecen en cero, mediante el siguiente fragmento de programa:

```
for ( int fila = 0; fila < ventas.length; fila++ )
{
    for ( int col = 0; col < ventas[ fila ].length; col++ )
    {
        ventas[ fila ][ col ] = 0;
    }
}
```

7.14 Escriba una aplicación que calcule el producto de una serie de enteros que se pasan al método `producto`, usando una lista de argumentos de longitud variable. Pruebe su método con varias llamadas, cada una con un número distinto de argumentos.

7.15 Modifique la figura 7.2, de manera que el tamaño del arreglo se especifique mediante el primer argumento de línea de comandos. Si no se suministra un argumento de línea de comandos, use 10 como el valor predeterminado del arreglo.

7.16 Escriba una aplicación que utilice una instrucción `for` mejorada para sumar los valores `double` que se pasan mediante los argumentos de línea de comandos. [*Sugerencia:* use el método `static parseDouble` de la clase `Double` para convertir un `String` en un valor `double`].

7.17 (*Tiro de dados*) Escriba una aplicación para simular el tiro de dos dados. La aplicación debe utilizar un objeto de la clase `Random` una vez para tirar el primer dado, y de nuevo para tirar el segundo dado. Después debe calcularse la suma de los dos valores. Cada dado puede mostrar un valor entero del 1 al 6, por lo que la suma de los valores variará del 2 al 12, siendo 7 la suma más frecuente, mientras que 2 y 12 serán las sumas menos frecuentes. En la figura 7.30 se muestran las 36 posibles combinaciones de los dos dados. Su aplicación debe tirar los dados 36,000 veces. Utilice un arreglo unidimensional para registrar el número de veces que aparezca cada una de las posibles sumas. Muestre los resultados en formato tabular. Determine si los totales son razonables (es decir, hay seis formas de tirar un 7, por lo que aproximadamente una sexta parte de los tiros deben ser 7).

7.18 (*Juego de craps*) Escriba una aplicación que ejecute 1000 juegos de craps (figura 6.9) y responda a las siguientes preguntas:

a) ¿Cuántos juegos se ganan en el primer tiro, en el segundo, …, en el vigésimo tiro y después de éste?
b) ¿Cuántos juegos se pierden en el primer tiro, en el segundo, …, en el vigésimo tiro y después de éste?
c) ¿Cuáles son las probabilidades de ganar en craps? [*Nota:* debe descubrir que craps es uno de los juegos de casino más justos. ¿Qué cree usted que significa esto?].
d) ¿Cuál es la duración promedio de un juego de craps?
e) ¿Las probabilidades de ganar mejoran con la duración del juego?

	1	2	3	4	5	6
1	2	3	4	5	6	7
2	3	4	5	6	7	8
3	4	5	6	7	8	9
4	5	6	7	8	9	10
5	6	7	8	9	10	11
6	7	8	9	10	11	12

Figura 7.30 | Las 36 posibles sumas de dos dados.

7.19 (*Sistema de reservaciones de una aerolínea)* Una pequeña aerolínea acaba de comprar una computadora para su nuevo sistema de reservaciones automatizado. Se le ha pedido a usted que desarrolle el nuevo sistema. Usted escribirá una aplicación para asignar asientos en cada vuelo del único avión de la aerolínea (capacidad: 10 asientos).

Su aplicación debe mostrar las siguientes alternativas: `Por favor escriba 1 para Primera Clase` y `Por favor escriba 2 para Economico`. Si el usuario escribe `1`, su aplicación debe asignarle un asiento en la sección de primera clase (asientos 1 a 5). Si el usuario escribe `2`, su aplicación debe asignarle un asiento en la sección económica (asientos 6 a 10). Su aplicación deberá entonces imprimir un pase de abordaje, indicando el número de asiento de la persona y si se encuentra en la sección de primera clase o clase económica del avión.

Use un arreglo unidimensional del tipo primitivo `boolean` para representar la tabla de asientos del avión. Inicialice todos los elementos del arreglo con `false` para indicar que todos los asientos están vacíos. A medida que se asigne cada asiento, establezca los elementos correspondientes del arreglo en `true` para indicar que ese asiento ya no está disponible.

Su aplicación nunca deberá asignar un asiento que ya haya sido asignado. Cuando esté llena la sección económica, su programa deberá preguntar a la persona si acepta ser colocada en la sección de primera clase (y viceversa). Si la persona acepta, haga la asignación de asiento apropiada. Si no acepta, imprima el mensaje `"El proximo vuelo sale en 3 horas"`.

7.20 (*Ventas totales*) Use un arreglo bidimensional para resolver el siguiente problema: una compañía tiene cuatro vendedores (1 a 4) que venden cinco productos distintos (1 a 5). Una vez al día, cada vendedor pasa una nota por cada tipo de producto vendido. Cada nota contiene lo siguiente:

a) El número del vendedor.
b) El número del producto.
c) El valor total en dólares de ese producto vendido en ese día.

Así, cada vendedor pasa entre 0 y 5 notas de venta por día. Suponga que está disponible la información sobre todas las notas del mes pasado. Escriba una aplicación que lea toda esta información para las ventas del último mes y que resuma las ventas totales por vendedor, por producto. Todos los totales deben guardarse en el arreglo bidimensional `ventas`. Después de procesar toda la información del mes pasado, muestre los resultados en formato tabular, en donde cada columna represente a un vendedor específico y cada fila represente a un producto. Saque el total de cada fila para obtener las ventas totales de cada producto durante el último mes. Saque el total de cada columna para obtener las ventas totales de cada vendedor durante el último mes. Su impresión tabular debe incluir estos totales cruzados a la derecha de las filas totalizadas, y en la parte inferior de las columnas totalizadas.

7.21 (*Gráficos de tortuga*) El lenguaje Logo hizo famoso el concepto de los *gráficos de tortuga*. Imagine a una tortuga mecánica que camina por todo el cuarto, bajo el control de una aplicación en Java. La tortuga sostiene una pluma en una de dos posiciones, arriba o abajo. Mientras la pluma está abajo, la tortuga va trazando figuras a medida que se va moviendo, y mientras la pluma está arriba, la tortuga se mueve alrededor libremente, sin trazar nada. En este problema usted simulará la operación de la tortuga y creará un bloc de dibujo computarizado.

Utilice un arreglo de 20 por 20 llamado `piso`, que se inicialice con ceros. Lea los comandos de un arreglo que los contenga. Lleve el registro de la posición actual de la tortuga en todo momento, y si la pluma se encuentra arriba o abajo. Suponga que la tortuga siempre empieza en la posición (0, 0) del piso, con su pluma hacia arriba. El conjunto de comandos de la tortuga que su aplicación debe procesar se muestra en la figura 7.31.

Suponga que la tortuga se encuentra en algún lado cerca del centro del piso. El siguiente “programa” dibuja e imprime un cuadrado de 12 por 12, dejando la pluma en posición levantada:

```
2
5,12
3
5,12
3
5,12
3
5,12
1
6
9
```

A medida que la tortuga se vaya desplazando con la pluma hacia abajo, asigne `1` a los elementos apropiados del arreglo `piso`. Cuando se dé el comando `6` (imprimir el arreglo), siempre que haya un `1` en el arreglo muestre un asterisco o cualquier carácter que usted elija. Siempre que haya un `0`, muestre un carácter en blanco.

Comando	Significado
`1`	Pluma arriba.
`2`	Pluma abajo.
`3`	Voltear a la derecha.
`4`	Voltear a la izquierda.
`5, 10`	Avanzar hacia delante 10 espacios (reemplace el 10 por un número distinto de espacios).
`6`	Imprimir el arreglo de 20 por 20.
`9`	Fin de los datos (centinela).

Figura 7.31 | Comandos de gráficos de tortuga.

Escriba una aplicación para implementar las herramientas de gráficos de tortuga aquí descritas. Escriba varios programas de gráficos de tortuga para dibujar figuras interesantes. Agregue otros comandos para incrementar el poder de su lenguaje de gráficos de tortuga.

7.22 (*Paseo del caballo*) Uno de los enigmas más interesantes para los entusiastas del ajedrez es el problema del Paseo del caballo, propuesto originalmente por el matemático Euler. ¿Puede la pieza de ajedrez, conocida como caballo, moverse alrededor de un tablero de ajedrez vacío y tocar cada una de las 64 posiciones una y sólo una vez? A continuación estudiaremos detalladamente este intrigante problema.

El caballo realiza solamente movimientos en forma de L (dos espacios en una dirección y un espacio en una dirección perpendicular). Por lo tanto, como se muestra en la figura 7.32, desde una posición cerca del centro de un tablero de ajedrez vacío, el caballo (etiquetado como `C`) puede hacer ocho movimientos distintos (numerados del 0 al 7).

a) Dibuje un tablero de ajedrez de ocho por ocho en una hoja de papel, e intente realizar un Paseo del caballo en forma manual. Ponga un `1` en la posición inicial, un `2` en la segunda posición, un `3` en la tercera, etcétera. Antes de empezar el paseo, estime qué tan lejos podrá avanzar, recordando que un paseo completo consta de 64 movimientos. ¿Qué tan lejos llegó? ¿Estuvo esto cerca de su estimación?

b) Ahora desarrollaremos una aplicación para mover el caballo alrededor de un tablero de ajedrez. El tablero estará representado por un arreglo bidimensional llamado `tablero`, de ocho por ocho. Cada posición se inicializará con cero. Describiremos cada uno de los ocho posibles movimientos en términos de sus componentes horizontales y verticales. Por ejemplo, un movimiento de tipo 0, como se muestra en la figura 7.32, consiste en mover dos posiciones horizontalmente a la derecha y una posición verticalmente hacia arriba. Un movimiento de tipo 2 consiste en mover una posición horizontalmente a la izquierda y dos posiciones verticalmente hacia arriba. Los movimientos horizontal a la izquierda y vertical hacia arriba se indican con números negativos. Los ocho movimientos pueden describirse mediante dos arreglos unidimensionales llamados `horizontal` y `vertical`, de la siguiente manera:

```
horizontal[ 0 ] = 2          vertical[ 0 ] = -1
horizontal[ 1 ] = 1          vertical[ 1 ] = -2
horizontal[ 2 ] = -1         vertical[ 2 ] = -2
horizontal[ 3 ] = -2         vertical[ 3 ] = -1
horizontal[ 4 ] = -2         vertical[ 4 ] = 1
horizontal[ 5 ] = -1         vertical[ 5 ] = 2
horizontal[ 6 ] = 1          vertical[ 6 ] = 2
horizontal[ 7 ] = 2          vertical[ 7 ] = 1
```

Deje que las variables `filaActual` y `columnaActual` indiquen la fila y columna, respectivamente, de la posición actual del caballo. Para hacer un movimiento de tipo `numeroMovimiento`, en donde `numeroMovimiento` puede estar entre 0 y 7, su programa debe utilizar las instrucciones

```
filaActual += vertical[ numeroMovimiento ];
columnaActual += horizontal[ numeroMovimiento];
```



Figura 7.32 | Los ocho posibles movimientos del caballo.

Escriba una aplicación para mover el caballo alrededor del tablero de ajedrez. Utilice un contador que varíe de `1` a `64`. Registre la última cuenta en cada posición a la que se mueva el caballo. Evalúe cada movimiento potencial para ver si el caballo ya visitó esa posición. Pruebe cada movimiento potencial para asegurarse que el caballo no se salga del tablero de ajedrez. Ejecute la aplicación. ¿Cuántos movimientos hizo el caballo?

c) Después de intentar escribir y ejecutar una aplicación de Paseo del caballo, probablemente haya desarrollado algunas ideas valiosas. Utilizaremos estas ideas para desarrollar una *heurística* (o "regla empírica") para mover el caballo. La heurística no garantiza el éxito, pero una heurística cuidadosamente desarrollada mejora de manera considerable la probabilidad de tener éxito. Probablemente usted ya observó que las posiciones externas son más difíciles que las posiciones cercanas al centro del tablero. De hecho, las posiciones más difíciles o inaccesibles son las cuatro esquinas.

La intuición sugiere que usted debe intentar mover primero el caballo a las posiciones más problemáticas y dejar pendientes aquellas a las que sea más fácil llegar, de manera que cuando el tablero se congestione cerca del final del paseo, habrá una mayor probabilidad de éxito.

Podríamos desarrollar una "heurística de accesibilidad" clasificando cada una de las posiciones de acuerdo a qué tan accesibles son y luego mover siempre el caballo (usando los movimientos en L) a la posición más inaccesible. Etiquetaremos un arreglo bidimensional llamado `accesibilidad` con números que indiquen desde cuántas posiciones es accesible una posición determinada. En un tablero de ajedrez en blanco, cada una de las 16 posiciones más cercanas al centro se clasifican con `8`; cada posición en la esquina se clasifica con `2`; y las demás posiciones tienen números de accesibilidad `3`, `4` o `6`, de la siguiente manera:

```
2  3  4  4  4  4  3  2
3  4  6  6  6  6  4  3
4  6  8  8  8  8  6  4
4  6  8  8  8  8  6  4
4  6  8  8  8  8  6  4
4  6  8  8  8  8  6  4
3  4  6  6  6  6  4  3
2  3  4  4  4  4  3  2
```

Escriba una nueva versión del Paseo del caballo, utilizando la heurística de accesibilidad. El caballo deberá moverse siempre a la posición con el número de accesibilidad más bajo. En caso de un empate, el caballo podrá moverse a cualquiera de las posiciones empatadas. Por lo tanto, el paseo puede empezar en cualquiera de las cuatro esquinas. [*Nota:* al ir moviendo el caballo alrededor del tablero, su aplicación deberá reducir los números de accesibilidad a medida que se vayan ocupando más posiciones. De esta manera y en cualquier momento dado durante el paseo, el número de accesibilidad de cada una de las posiciones disponibles seguirá siendo igual al número preciso de posiciones desde las que se puede llegar a esa posición]. Ejecute esta versión de su aplicación. ¿Logró completar el paseo? Modifique el programa para realizar 64 paseos, en donde cada uno empiece desde una posición distinta en el tablero. ¿Cuántos paseos completos logró realizar?

d) Escriba una versión del programa del Paseo del caballo que, al encontrarse con un empate entre dos o más posiciones, decida qué posición elegir buscando más adelante aquellas posiciones que se puedan alcanzar desde las posiciones "empatadas". Su aplicación debe mover el caballo a la posición empatada para la cual el siguiente movimiento lo lleve a una posición con el número de accesibilidad más bajo.

7.23 (*Paseo del caballo: métodos de fuerza bruta*) En la parte (c) del ejercicio 7.22, desarrollamos una solución al problema del Paseo del caballo. El método utilizado, llamado "heurística de accesibilidad", genera muchas soluciones y se ejecuta con eficiencia.

A medida que se incremente de manera continua la potencia de las computadoras, seremos capaces de resolver más problemas con menos potencia y algoritmos relativamente menos sofisticados. A éste le podemos llamar el método de la "fuerza bruta" para resolver problemas.

a) Utilice la generación de números aleatorios para permitir que el caballo se desplace a lo largo del tablero (mediante sus movimientos legítimos en L) en forma aleatoria. Su aplicación debe ejecutar un paseo e imprimir el tablero final. ¿Qué tan lejos llegó el caballo?

b) La mayoría de las veces, la aplicación de la parte (a) produce un paseo relativamente corto. Ahora modifique su aplicación para intentar 1000 paseos. Use un arreglo unidimensional para llevar el registro del número

de paseos de cada longitud. Cuando su programa termine de intentar los 1000 paseos, deberá imprimir esta información en un formato tabular ordenado. ¿Cuál fue el mejor resultado?

c) Es muy probable que la aplicación de la parte (b) le haya brindado algunos paseos "respetables", pero no completos. Ahora deje que su aplicación se ejecute hasta que produzca un paseo completo. [*Precaución:* esta versión del programa podría ejecutarse durante horas en una computadora poderosa]. Una vez más, mantenga una tabla del número de paseos de cada longitud e imprímala cuando se encuentre el primer paseo completo. ¿Cuántos paseos intentó su programa antes de producir uno completo? ¿Cuánto tiempo se tomó?

d) Compare la versión de la fuerza bruta del Paseo del caballo con la versión heurística de accesibilidad. ¿Cuál requirió un estudio más cuidadoso del problema? ¿Qué algoritmo fue más difícil de desarrollar? ¿Cuál requirió más poder de cómputo? ¿Podríamos tener la certeza (por adelantado) de obtener un paseo completo mediante el método de la heurística de accesibilidad? ¿Podríamos tener la certeza (por adelantado) de obtener un paseo completo mediante el método de la fuerza bruta? Argumente las ventajas y desventajas de solucionar el problema mediante la fuerza bruta en general.

7.24 (*Ocho reinas*) Otro enigma para los entusiastas del ajedrez es el problema de las Ocho reinas, el cual pregunta lo siguiente: ¿es posible colocar ocho reinas en un tablero de ajedrez vacío, de tal manera que ninguna "ataque" a cualquier otra (es decir, que no haya dos reinas en la misma fila, en la misma columna o a lo largo de la misma diagonal)? Use la idea desarrollada en el ejercicio 7.22 para formular una heurística para resolver el problema de las Ocho reinas. Ejecute su aplicación. [*Sugerencia:* es posible asignar un valor a cada una de las posiciones en el tablero de ajedrez, para indicar cuántas posiciones de un tablero vacío se "eliminan" si una reina se coloca en esa posición. A cada una de las esquinas se le asignaría el valor 22, como se demuestra en la figura 7.33. Una vez que estos "números de eliminación" se coloquen en las 64 posiciones, una heurística apropiada podría ser: coloque la siguiente reina en la posición con el número de eliminación más pequeño. ¿Por qué esta estrategia es intuitivamente atractiva?].

7.25 (*Ocho reinas: métodos de fuerza bruta*) En este ejercicio usted desarrollará varios métodos de fuerza bruta para resolver el problema de las Ocho reinas que presentamos en el ejercicio 7.24.

a) Utilice la técnica de la fuerza bruta aleatoria desarrollada en el ejercicio 7.23, para resolver el problema de las Ocho reinas.

b) Utilice una técnica exhaustiva (es decir, pruebe todas las combinaciones posibles de las ocho reinas en el tablero) para resolver el problema de las Ocho reinas.

c) ¿Por qué el método de la fuerza bruta exhaustiva podría no ser apropiado para resolver el problema del Paseo del caballo?

d) Compare y contraste el método de la fuerza bruta aleatoria con el de la fuerza bruta exhaustiva.

7.26 (*Paseo del caballo: prueba del paseo cerrado*) En el Paseo del caballo (ejercicio 7.22), se lleva a cabo un paseo completo cuando el caballo hace 64 movimientos, en los que toca cada esquina del tablero una sola vez. Un paseo cerrado ocurre cuando el movimiento 64 se encuentra a un movimiento de distancia de la posición en la que el caballo empezó el paseo. Modifique el programa que escribió en el ejercicio 7.22 para probar si el paseo ha sido completo, y si se trató de un paseo cerrado.

7.27 (*La criba de Eratóstenes*) Un número primo es cualquier entero mayor que 1, divisible sólo por sí mismo y por el número 1. La Criba de Eratóstenes es un método para encontrar números primos, el cual opera de la siguiente manera:

*	*	*	*	*	*	*	*
*	*						
*		*					
*			*				
*				*			
*					*		
*						*	
*							*

Figura 7.33 | Las 22 posiciones eliminadas al colocar una reina en la esquina superior izquierda.

a) Cree un arreglo del tipo primitivo `boolean`, con todos los elementos inicializados en `true`. Los elementos del arreglo con índices primos permanecerán como `true`. Cualquier otro elemento del arreglo eventualmente cambiará a `false`.
b) Empezando con el índice 2 del arreglo, determine si un elemento dado es `true`. De ser así, itere a través del resto del arreglo y asigne `false` a todo elemento cuyo índice sea múltiplo del índice del elemento que tiene el valor `true`. Después continúe el proceso con el siguiente elemento que tenga el valor `true`. Para el índice 2 del arreglo, todos los elementos más allá del elemento 2 en el arreglo que tengan índices múltiplos de 2 (los índices 4, 6, 8, 10, etcétera) se establecerán en `false`; para el índice 3 del arreglo, todos los elementos más allá del elemento 3 en el arreglo que tengan índices múltiplos de 3 (los índices 6, 9, 12, 15, etcétera) se establecerán en `false`; y así sucesivamente.

Cuando este proceso termine, los elementos del arreglo que aún sean `true` indicarán que el índice es un número primo. Estos índices pueden mostrarse. Escriba una aplicación que utilice un arreglo de 1000 elementos para determinar e imprimir los números primos entre 2 y 999. Ignore los elementos 0 y 1 del arreglo.

7.28 (*Simulación: la tortuga y la liebre*) En este problema usted recreará la clásica carrera de la tortuga y la liebre. Utilizará la generación de números aleatorios para desarrollar una simulación de este memorable suceso.

Nuestros competidores empezarán la carrera en la posición 1 de 70 posiciones. Cada posición representa a una posible posición a lo largo del curso de la carrera. La línea de meta se encuentra en la posición 70. El primer competidor en llegar a la posición 70 recibirá una cubeta llena con zanahorias y lechuga frescas. El recorrido se abre paso hasta la cima de una resbalosa montaña, por lo que ocasionalmente los competidores pierden terreno.

Un reloj hace tictac una vez por segundo. Con cada tic del reloj, su aplicación debe ajustar la posición de los animales de acuerdo con las reglas de la figura 7.34. Use variables para llevar el registro de las posiciones de los animales (los números son del 1 al 70). Empiece con cada animal en la posición 1 (la "puerta de inicio"). Si un animal se resbala hacia la izquierda antes de la posición 1, regréselo a la posición 1.

Genere los porcentajes en la figura 7.34 produciendo un entero aleatorio i en el rango $1 \leq i \leq 10$. Para la tortuga, realice un "paso pesado rápido" cuando $1 \leq i \leq 5$, un "resbalón" cuando $6 \leq i \leq 7$ o un "paso pesado lento" cuando $8 \leq i \leq 10$. Utilice una técnica similar para mover a la liebre.

Empiece la carrera imprimiendo el mensaje

```
PUM!!!
Y ARRANCAN!!!
```

Luego, para cada tic del reloj (es decir, cada repetición de un ciclo) imprima una línea de 70 posiciones, mostrando la letra `T` en la posición de la tortuga y la letra `H` en la posición de la liebre. En ocasiones los competidores se encontrarán en la misma posición. En este caso, la tortuga muerde a la liebre y su aplicación debe imprimir `OUCH!!!` empezando en esa posición. Todas las posiciones de impresión distintas de la `T`, la `H` o el mensaje `OUCH!!!` (en caso de un empate) deben estar en blanco.

Animal	Tipo de movimiento	Porcentaje del tiempo	Movimiento actual
Tortuga	Paso pesado rápido	50%	3 posiciones a la derecha
	Resbalón	20%	6 posiciones a la izquierda
	Paso pesado lento	30%	1 posición a la derecha
Liebre	Dormir	20%	Ningún movimiento
	Gran salto	20%	9 posiciones a la derecha
	Gran resbalón	10%	12 posiciones a la izquierda
	Pequeño salto	30%	1 posición a la derecha
	Pequeño resbalón	20%	2 posiciones a la izquierda

Figura 7.34 | Reglas para ajustar las posiciones de la tortuga y la liebre.

Después de imprimir cada línea, compruebe si uno de los animales ha llegado o se ha pasado de la posición 70. De ser así, imprima quién fue el ganador y termine la simulación. Si la tortuga gana, imprima `LA TORTUGA GANA!!! YAY!!!` Si la liebre gana, imprima `La liebre gana. Que mal.` Si ambos animales ganan en el mismo tic del reloj, tal vez usted quiera favorecer a la tortuga (la más débil) o tal vez quiera imprimir `Es un empate.` Si ninguno de los dos animales gana, ejecute el ciclo de nuevo para simular el siguiente tic del reloj. Cuando esté listo para ejecutar su aplicación, reúna a un grupo de aficionados para que vean la carrera. ¡Se sorprenderá al ver lo participativa que puede ser su audiencia!

Posteriormente presentaremos una variedad de herramientas de Java, como gráficos, imágenes, animación, sonido y subprocesamiento múltiple. Cuando estudie esas herramientas, tal vez pueda disfrutar mejorando su simulación de la tortuga y la liebre.

7.29 (*Serie de Fibonacci*) La serie de Fibonacci

0, 1, 1, 2, 3, 5, 8, 13, 21, ...

empieza con los términos 0 y 1, y tiene la propiedad de que cada término sucesivo es la suma de los dos términos anteriores.

a) Escriba un método llamado `fibonacci( n )` que calcule el *enésimo* número de Fibonacci. Incorpore este método en una aplicación que permita al usuario introducir el valor de `n`.
b) Determine el número de Fibonacci más grande que puede imprimirse en su sistema.
c) Modifique la aplicación que escribió en la parte (a), de manera que utilice `double` en vez de `int` para calcular y devolver números de Fibonacci, y utilice esta aplicación modificada para repetir la parte (b).

Los ejercicios 7.30 a 7.33 son de una complejidad razonable. Una vez que haya resuelto estos problemas, obtendrá la capacidad de implementar la mayoría de los juegos populares de cartas con facilidad.

7.30 *(Barajar y repartir cartas)* Modifique la aplicación de la figura 7.11 para repartir una mano de póquer de cinco cartas. Después modifique la clase `PaqueteDeCartas` de la figura 7.10 para incluir métodos que determinen si una mano contiene

a) un par
b) dos pares
c) tres de un mismo tipo (como tres jotos)
d) cuatro de un mismo tipo (como cuatro ases)
e) una corrida (es decir, las cinco cartas del mismo palo)
f) una escalera (es decir, cinco cartas de valor consecutivo de la misma cara)
g) "full house" (es decir, dos cartas de un valor de la misma cara y tres cartas de otro valor de la misma cara)

[*Sugerencia:* agregue los métodos `obtenerCara` y `obtenerPalo` a la clase `Carta` de la figura 7.9.]

7.31 *(Barajar y repartir cartas)* Use los métodos desarrollados en el ejercicio 7.30 para escribir una aplicación que reparta dos manos de póquer de cinco cartas, que evalúe cada mano y determine cuál de las dos es mejor.

7.32 *(Barajar y repartir cartas)* Modifique la aplicación desarrollada en el ejercicio 7.31, de manera que pueda simular el repartidor. La mano de cinco cartas del repartidor se reparte "cara abajo", por lo que el jugador no puede verla. A continuación, la aplicación debe evaluar la mano del repartidor y, con base en la calidad de ésta, debe sacar una, dos o tres cartas más para reemplazar el número correspondiente de cartas que no necesita en la mano original. Después, la aplicación debe reevaluar la mano del repartidor. [*Precaución:* ¡éste es un problema difícil!].

7.33 *(Barajar y repartir cartas)* Modifique la aplicación desarrollada en el ejercicio 7.32, de manera que pueda encargarse de la mano del repartidor automáticamente, pero debe permitir al jugador decidir cuáles cartas de su mano desea reemplazar. A continuación, la aplicación deberá evaluar ambas manos y determinar quién gana. Ahora utilice esta nueva aplicación para jugar 20 manos contra la computadora. ¿Quién gana más juegos, usted o la computadora? Haga que un amigo juegue 20 manos contra la computadora. ¿Quién gana más juegos? Con base en los resultados de estos juegos, refine su aplicación para jugar póquer. (Esto también es un problema difícil). Juegue 20 manos más. ¿Su aplicación modificada hace un mejor juego?

Sección especial: construya su propia computadora

En los siguientes problemas nos desviaremos temporalmente del mundo de la programación en lenguajes de alto nivel, para "abrir de par en par" una computadora y ver su estructura interna. Presentaremos la programación en lenguaje

máquina y escribiremos varios programas en este lenguaje. Para que ésta sea una experiencia valiosa, crearemos también una computadora (mediante la técnica de la *simulación* basada en software) en la que pueda ejecutar sus programas en lenguaje máquina.

7.34 (*Programación en lenguaje máquina*) Crearemos una computadora a la que llamaremos Simpletron. Como su nombre lo indica, es una máquina simple, pero poderosa. Simpletron sólo ejecuta programas escritos en el único lenguaje que entiende directamente: el lenguaje máquina de Simpletron , o LMS.

Simpletron contiene un *acumulador*, un registro especial en el cual se coloca la información antes de que Simpletron la utilice en los cálculos, o que la analice de distintas maneras. Toda la información dentro de Simpletron se manipula en términos de *palabras*. Una palabra es un número decimal con signo de cuatro dígitos, como `+3364`, `-1293`, `+0007` y `-0001`. Simpletron está equipada con una memoria de 100 palabras, y se hace referencia a estas palabras mediante sus números de ubicación `00`, `01`, ..., `99`.

Antes de ejecutar un programa LMS debemos *cargar*, o colocar, el programa en memoria. La primera instrucción de cada programa LMS se coloca siempre en la ubicación `00`. El simulador empezará a ejecutarse en esta ubicación.

Cada instrucción escrita en LMS ocupa una palabra de la memoria de Simpletron (y, por lo tanto, las instrucciones son números decimales de cuatro dígitos con signo). Supondremos que el signo de una instrucción LMS siempre será positivo, pero el signo de una palabra de información puede ser positivo o negativo. Cada una de las ubicaciones en la memoria de Simpletron puede contener una instrucción, un valor de datos utilizado por un programa o un área no utilizada (y, por lo tanto, indefinida) de memoria. Los primeros dos dígitos de cada instrucción LMS son el *código de operación* que especifica la operación a realizar. Los códigos de operación de LMS se sintetizan en la figura 7.35.

Los últimos dos dígitos de una instrucción LMS son el *operando* (la dirección de la ubicación en memoria que contiene la palabra a la cual se aplica la operación). Consideremos varios programas simples en LMS.

Código de operación	Significado
Operaciones de entrada/salida:	
`final int LEE = 10;`	Lee una palabra desde el teclado y la introduce en una ubicación específica de memoria.
`final int ESCRIBE = 11;`	Escribe una palabra de una ubicación específica de memoria y la imprime en la pantalla.
Operaciones de carga/almacenamiento:	
`final int CARGA = 20;`	Carga una palabra de una ubicación específica de memoria y la coloca en el acumulador.
`final int ALMACENA = 21;`	Almacena una palabra del acumulador dentro de una ubicación específica de memoria.
Operaciones aritméticas:	
`final int SUMA = 30;`	Suma una palabra de una ubicación específica de memoria a la palabra en el acumulador (deja el resultado en el acumulador).
`final int RESTA = 31;`	Resta una palabra de una ubicación específica de memoria a la palabra en el acumulador (deja el resultado en el acumulador).
`final int DIVIDE = 32;`	Divide una palabra de una ubicación específica de memoria entre la palabra en el acumulador (deja el resultado en el acumulador).
`final int MULTIPLICA = 33;`	Multiplica una palabra de una ubicación específica de memoria por la palabra en el acumulador (deja el resultado en el acumulador).
Operaciones de transferencia de control:	
`final int BIFURCA = 40;`	Bifurca hacia una ubicación específica de memoria.

Figura 7.35 | Códigos de operación del Lenguaje máquina Simpletron (LMS). (Parte 1 de 2).

Código de operación	Significado
Operaciones de transferencia de control:	
`final int BIFURCANEG = 41;`	Bifurca hacia una ubicación específica de memoria si el acumulador es negativo.
`final int BIFURCACERO = 42;`	Bifurca hacia una ubicación específica de memoria si el acumulador es cero.
`const int ALTO = 43;`	Alto. El programa completó su tarea.

Figura 7.35 | Códigos de operación del Lenguaje máquina Simpletron (LMS). (Parte 2 de 2).

El primer programa en LMS (figura 7.36) lee dos números del teclado, calcula e imprime su suma. La instrucción `+1007` lee el primer número del teclado y lo coloca en la ubicación `07` (que se ha inicializado con 0). Después, la instrucción `+1008` lee el siguiente número y lo coloca en la ubicación `08`. La instrucción *carga*, `+2007`, coloca el primer número en el acumulador y la instrucción *suma*, `+3008`, suma el segundo número al número en el acumulador. *Todas las instrucciones LMS aritméticas dejan sus resultados en el acumulador*. La instrucción *almacena*, `+2109`, coloca el resultado de vuelta en la ubicación de memoria `09`, desde la cual la instrucción *escribe*, `+1109`, toma el número y lo imprime (como un número decimal de cuatro dígitos con signo). La instrucción *alto*, `+4300`, termina la ejecución.

Ubicación	Número	Instrucción
00	+1007	(Lee A)
01	+1008	(Lee B)
02	+2007	(Carga A)
03	+3008	(Suma B)
04	+2109	(Almacena C)
05	+1109	(Escribe C)
06	+4300	(Alto)
07	+0000	(Variable A)
08	+0000	(Variable B)
09	+0000	(Resultado C)

Figura 7.36 | Programa en LMS que lee dos enteros y calcula la suma.

El segundo programa en LMS (figura 7.37) lee dos números desde el teclado, determina e imprime el valor más grande. Observe el uso de la instrucción `+4107` como una transferencia de control condicional, en forma muy similar a la instrucción `if` de Java.

Ubicación	Número	Instrucción
00	+1009	(Lee A)
01	+1010	(Lee B)

Figura 7.37 | Programa en LMS que lee dos enteros y determina cuál de ellos es mayor. (Parte 1 de 2).

Ubicación	Número	Instrucción
02	+2009	(Carga A)
03	+3110	(Resta B)
04	+4107	(Bifurcación negativa a 07)
05	+1109	(Escribe A)
06	+4300	(Alto)
07	+1110	(Escribe B)
08	+4300	(Alto)
09	+0000	(Variable A)
10	+0000	(Variable B)

Figura 7.37 | Programa en LMS que lee dos enteros y determina cuál de ellos es mayor. (Parte 2 de 2).

Ahora escriba programas en LMS para realizar cada una de las siguientes tareas:

a) Usar un ciclo controlado por centinela para leer 10 números positivos. Calcular e imprimir la suma.
b) Usar un ciclo controlado por contador para leer siete números, algunos positivos y otros negativos, y calcular e imprimir su promedio.
c) Leer una serie de números, determinar e imprimir el número más grande. El primer número leído indica cuántos números deben procesarse.

7.35 (*Un simulador de computadora*) En este problema usted creará su propia computadora. No, no soldará componentes, sino que utilizará la poderosa técnica de la *simulación basada en software* para crear un *modelo de software* orientado a objetos de Simpletron, la computadora del ejercicio 7.34. Su simulador Simpletron convertirá la computadora que usted utiliza en Simpletron, y será capaz de ejecutar, probar y depurar los programas LMS que escribió en el ejercicio 7.34.

Cuando ejecute su simulador Simpletron, debe empezar mostrando lo siguiente:

```
*** Bienvenido a Simpletron! ***
*** Por favor, introduzca en su programa una instruccion   ***
*** (o palabra de datos) a la vez en el campo de texto de  ***
*** entrada. Yo le mostrare el numero de ubicacion y       ***
*** un signo de interrogacion (?). Entonces usted          ***
*** escribira la palabra para esa ubicacion. Oprima el     ***
*** boton Terminar para dejar de introducir su programa.   ***
```

Su aplicación debe simular la memoria del Simpletron con un arreglo unidimensional llamado `memoria`, que cuente con 100 elementos. Ahora suponga que el simulador se está ejecutando y examinaremos el diálogo a medida que introduzcamos el programa de la figura 7.37 (ejercicio 7.34):

```
00 ? +1009
01 ? +1010
02 ? +2009
03 ? +3110
04 ? +4107
05 ? +1109
06 ? +4300
07 ? +1110
08 ? +4300
09 ? +0000
10 ? +0000
11 ? -99999
```

Su programa debe mostrar la ubicación en memoria, seguida por un signo de interrogación. Cada uno de los valores a la derecha de un signo de interrogación es introducido por el usuario. Al introducir el valor centinela `-99999`, el programa debe mostrar lo siguiente:

```
*** Se completo la carga del programa  ***
*** Empieza la ejecucion del programa ***
```

Ahora el programa en LMS se ha colocado (o cargado) en el arreglo `memoria`. Simpletron debe a continuación ejecutar el programa en LMS. La ejecución comienza con la instrucción en la ubicación `00` y, como en Java, continúa secuencialmente a menos que se lleve a otra parte del programa mediante una transferencia de control.

Use la variable `acumulador` para representar el registro acumulador. Use la variable `contadorDeInstrucciones` para llevar el registro de la ubicación en memoria que contiene la instrucción que se está ejecutando. Use la variable `codigoDeOperacion` para indicar la operación que se esté realizando actualmente (es decir, los dos dígitos a la izquierda en la palabra de instrucción). Use la variable `operando` para indicar la ubicación de memoria en la que operará la instrucción actual. Por lo tanto, `operando` está compuesta por los dos dígitos más a la derecha de la instrucción que se esté ejecutando en esos momentos. No ejecute las instrucciones directamente desde la memoria. En vez de eso, transfiera la siguiente instrucción a ejecutar desde la memoria hasta una variable llamada `registroDeInstruccion`. Luego "recoja" los dos dígitos a la izquierda y colóquelos en `codigoDeOperacion`, después "recoja" los dos dígitos a la derecha y colóquelos en `operando`. Cuando Simpletron comience con la ejecución, todos los registros especiales se deben inicializar con cero.

Ahora vamos a "dar un paseo" por la ejecución de la primera instrucción LMS, `+1009` en la ubicación de memoria `00`. A este procedimiento se le conoce como *ciclo de ejecución de una instrucción*.

El `contadorDeInstrucciones` nos indica la ubicación de la siguiente instrucción a ejecutar. Nosotros *buscamos* el contenido de esa ubicación de `memoria`, utilizando la siguiente instrucción de Java:

```
registroDeInstruccion = memoria[ contadorDeInstrucciones ];
```

El código de operación y el operando se extraen del registro de instrucción, mediante las instrucciones

```
codigoDeOperacion = registroDeInstruccion / 100;
operando = registroDeInstruccion % 100;
```

Ahora, Simpletron debe determinar que el código de operación es en realidad un *lee* (en comparación con un *escribe*, *carga*, etcétera). Una instrucción `switch` establece la diferencia entre las 12 operaciones de LMS. En la instrucción `switch` se simula el comportamiento de varias instrucciones LMS, como se muestra en la figura 7.38. En breve hablaremos sobre las instrucciones de bifurcación y dejaremos las otras a usted.

Cuando el programa en LMS termine de ejecutarse, deberán mostrarse el nombre y contenido de cada registro, así como el contenido completo de la memoria. A este tipo de impresión se le denomina vaciado de la computadora (no, un vaciado de computadora no es un lugar al que van las computadoras viejas). Para ayudarlo a programar su método de vaciado, en la figura 7.39 se muestra un formato de vaciado de muestra. Observe que un vaciado, después de la ejecución de un programa de Simpletron, muestra los valores actuales de las instrucciones y los valores de los datos al momento en que se terminó la ejecución.

Procedamos ahora con la ejecución de la primera instrucción de nuestro programa, `+1009` en la ubicación `00`. Como lo hemos indicado, la instrucción `switch` simula esta tarea pidiendo al usuario que escriba un valor, leyendo el valor y almacenándolo en la ubicación de memoria `memoria[ operando ]`. A continuación, el valor se lee y se coloca en la ubicación `09`.

Instrucción	Descripción
lee:	Mostrar el mensaje "`Escriba un entero`", después recibir como entrada el entero y almacenarlo en la ubicación `memoria[ operando ]`.
carga:	`acumulador = memoria[ operando ];`
suma:	`acumulador += memoria[ operando ];`
alto:	Esta instrucción muestra el mensaje `*** Termino la ejecución de Simpletron ***`

Figura 7.38 | Comportamiento de varias instrucciones de LMS en Simpletron.

```
REGISTROS
acumulador                    +0000
contadorDeInstrucciones          00
registroDeInstruccion         +0000
codigoDeOperacion                00
operando                         00

MEMORIA:

            0      1      2      3      4      5      6      7      8      9
    0   +0000  +0000  +0000  +0000  +0000  +0000  +0000  +0000  +0000  +0000
   10   +0000  +0000  +0000  +0000  +0000  +0000  +0000  +0000  +0000  +0000
   20   +0000  +0000  +0000  +0000  +0000  +0000  +0000  +0000  +0000  +0000
   30   +0000  +0000  +0000  +0000  +0000  +0000  +0000  +0000  +0000  +0000
   40   +0000  +0000  +0000  +0000  +0000  +0000  +0000  +0000  +0000  +0000
   50   +0000  +0000  +0000  +0000  +0000  +0000  +0000  +0000  +0000  +0000
   60   +0000  +0000  +0000  +0000  +0000  +0000  +0000  +0000  +0000  +0000
   70   +0000  +0000  +0000  +0000  +0000  +0000  +0000  +0000  +0000  +0000
   80   +0000  +0000  +0000  +0000  +0000  +0000  +0000  +0000  +0000  +0000
   90   +0000  +0000  +0000  +0000  +0000  +0000  +0000  +0000  +0000  +0000
```

Figura 7.39 Un vaciado de muestra.

En este punto se ha completado la simulación de la primera instrucción. Todo lo que resta es preparar a Simpletron para que ejecute la siguiente instrucción. Como la instrucción que acaba de ejecutarse no es una transferencia de control, sólo necesitamos incrementar el registro contador de instrucciones de la siguiente manera:

```
++contadorDeInstrucciones;
```

Esta acción completa la ejecución simulada de la primera instrucción. Todo el proceso (es decir, el ciclo de ejecución de una instrucción) empieza de nuevo, con la búsqueda de la siguiente instrucción a ser ejecutada.

Ahora veremos cómo se simulan las instrucciones de bifurcación (las transferencias de control). Todo lo que necesitamos hacer es ajustar el valor en el contador de instrucciones de manera apropiada. Por lo tanto, la instrucción de bifurcación condicional (40) se simula dentro de la instrucción `switch` como

```
contadorDeInstrucciones = operando;
```

La instrucción condicional "bifurcar si el acumulador es cero" se simula como

```
if ( acumulador == 0 )
    contadorDeInstrucciones = operando;
```

En este punto, usted debe implementar su simulador Simpletron y ejecutar cada uno de los programas que escribió en el ejercicio 7.34. Si lo desea, puede embellecer al LMS con características adicionales y ofrecerlas en su simulador.

Su simulador debe comprobar diversos tipos de errores. Por ejemplo, durante la fase de carga del programa, cada número que el usuario escribe en la `memoria` de Simpletron debe encontrarse dentro del rango de -9999 a +9999. Su simulador debe probar que cada número introducido se encuentre dentro de este rango y, en caso contrario, seguir pidiendo al usuario que vuelva a introducir el número hasta que introduzca un número correcto.

Durante la fase de ejecución, su simulador debe comprobar varios errores graves, como los intentos de dividir entre cero, intentos de ejecutar códigos de operación inválidos, y desbordamientos del acumulador (es decir, las operaciones aritméticas que den como resultado valores mayores que +9999 o menores que -9999). Dichos errores graves se conocen como *errores fatales*. Al detectar un error fatal, su simulador deberá imprimir un mensaje de error tal como

```
*** Intento de dividir entre cero ***
*** La ejecucion de Simpletron se termino en forma anormal ***
```

y deberá imprimir un vaciado de computadora completo en el formato que vimos anteriormente. Este análisis ayudará al usuario a localizar el error en el programa.

7.36 (*Modificaciones al simulador Simpletron*) En el ejercicio 7.35 usted escribió una simulación de software de una computadora que ejecuta programas escritos en el Lenguaje Máquina Simpletron (LMS). En este ejercicio proponemos varias modificaciones y mejoras al simulador Simpletron. En los ejercicios 17.26 y 17.27 propondremos la creación de un compilador que convierta los programas escritos en un lenguaje de programación de alto nivel (una variación de Basic) a Lenguaje Máquina Simpletron. Algunas de las siguientes modificaciones y mejoras pueden requerirse para ejecutar los programas producidos por el compilador:

a) Extienda la memoria del simulador Simpletron, de manera que contenga 1000 ubicaciones de memoria para permitir a Simpletron manejar programas más grandes.
b) Permita al simulador realizar cálculos de residuo. Esta modificación requiere de una instrucción adicional en LMS.
c) Permita al simulador realizar cálculos de exponenciación. Esta modificación requiere una instrucción adicional en LMS.
d) Modifique el simulador para que pueda utilizar valores hexadecimales, en vez de valores enteros para representar instrucciones en LMS.
e) Modifique el simulador para permitir la impresión de una nueva línea. Esta modificación requiere una instrucción adicional en LMS.
f) Modifique el simulador para procesar valores de punto flotante además de valores enteros.
g) Modifique el simulador para manejar la introducción de cadenas. [*Sugerencia:* cada palabra de Simpletron puede dividirse en dos grupos, cada una de las cuales guarda un entero de dos dígitos. Cada entero de dos dígitos representa el equivalente decimal de código ASCII (vea el apéndice B) de un carácter. Agregue una instrucción de lenguaje máquina que reciba como entrada una cadena y la almacene, empezando en una ubicación de memoria específica de Simpletron. La primera mitad de la palabra en esa ubicación será una cuenta del número de caracteres en la cadena (es decir, la longitud de la cadena). Cada media palabra subsiguiente contiene un carácter ASCII, expresado como dos dígitos decimales. La instrucción en lenguaje máquina convierte cada carácter en su equivalente ASCII y lo asigna a una media palabra].
h) Modifique el simulador para manejar la impresión de cadenas almacenadas en el formato de la parte (g). [*Sugerencia:* agregue una instrucción en lenguaje máquina que imprima una cadena, empezando en cierta ubicación de memoria de Simpletron. La primera mitad de la palabra en esa ubicación es una cuenta del número de caracteres en la cadena (es decir, la longitud de la misma). Cada media palabra subsiguiente contiene un carácter ASCII expresado como dos dígitos decimales. La instrucción en lenguaje máquina comprueba la longitud e imprime la cadena, traduciendo cada número de dos dígitos en su carácter equivalente].

8

Clases y objetos: un análisis más detallado



En vez de esta absurda división entre sexos, deberían clasificar a las personas como estáticas y dinámicas.

—Evelyn Waugh

¿Es éste un mundo en el cual se deben ocultar las virtudes?

—William Shakespeare

¿Pero qué cosa, para servir a nuestros fines privados, olvida los engaños de nuestros amigos?

—Charles Churchill

Por encima de todo: hay que ser sinceros con nosotros mismos.

—William Shakespeare

No hay que ser "duros", sino simplemente sinceros.

—Oliver Wendell Holmes, Jr.

OBJETIVOS

En este capítulo aprenderá a:

- Comprender el concepto de encapsulamiento y ocultamiento de datos.
- Comprender las nociones de la abstracción de datos y los tipos de datos abstractos (ADTs).
- Comprender el uso de la palabra clave `this`.
- Utilizar las variables y métodos `static`.
- Importar los miembros `static` de una clase.
- Utilizar el tipo `enum` para crear conjuntos de constantes con identificadores únicos.
- Declarar constantes `enum` con parámetros.
- Organizar las clases en paquetes para promover la reutilización.

Plan general

8.1 Introducción

En nuestras discusiones acerca de los programas orientados a objetos en los capítulos anteriores, presentamos muchos conceptos básicos y terminología en relación con la programación orientada a objetos (POO) en Java. También hablamos sobre nuestra metodología para desarrollar programas: seleccionamos variables y métodos apropiados para cada programa y especificamos la manera en la que un objeto de nuestra clase debería colaborar con los objetos de las clases en la API de Java para realizar los objetivos generales de la aplicación.

En este capítulo analizaremos más de cerca la creación de clases, el control del acceso a los miembros de una clase y la creación de constructores. Hablaremos sobre la composición: una capacidad que permite a una clase tener referencias a objetos de otras clases como miembros. Analizaremos nuevamente el uso de los métodos *establecer* y *obtener*, y exploraremos con más detalle el tipo de clase `enum` (presentado en la sección 6.10), el cual permite a los programadores declarar y manipular conjuntos de identificadores únicos, que representen valores constantes. En la sección 6.10 presentamos el tipo básico `enum`, el cual apareció dentro de otra clase y simplemente declaraba un conjunto de constantes. En este capítulo, hablaremos sobre la relación entre los tipos `enum` y las clases, demostrando que, al igual que una clase, un `enum` se puede declarar en su propio archivo con constructores, métodos y campos. El capítulo también habla detalladamente sobre los miembros de clase `static` y las variables de instancia `final`. Investigaremos cuestiones como la reutilización de software, la abstracción de datos y el encapsulamiento. Por último, explicaremos cómo organizar las clases en paquetes, para ayudar en la administración de aplicaciones extensas y promover la reutilización; después mostraremos una relación especial entre clases dentro del mismo paquete.

En el capítulo 9, Programación orientada a objetos: herencia, y en el capítulo 10, Programación orientada a objetos: polimorfismo, presentaremos dos tecnologías clave adicionales de la programación orientada a objetos.

8.2 Ejemplo práctico de la clase Tiempo

Declaración de la clase Tiempo1

Nuestro primer ejemplo consiste en dos clases: Tiempo1 (figura 8.1) y PruebaTiempo1 (figura 8.2). La clase Tiempo1 representa la hora del día. La clase PruebaTiempo1 es una clase de aplicación en la que el método main crea un objeto de la clase Tiempo1 e invoca a sus métodos. Estas clases se deben declarar en filas separadas ya que ambas son de tipo public. El resultado de este programa aparece en la figura 8.2.

La clase Tiempo1 contiene tres variables de instancia private de tipo int (figura 8.1, líneas 6 a 8): hora, minuto y segundo, que representan la hora en formato de tiempo universal (formato de reloj de 24 horas, en el cual las horas se encuentran en el rango de 0 a 23). La clase Tiempo1 contiene los métodos public establecerTiempo (líneas 12 a 17), aStringUniversal (líneas 20 a 23) y toString (líneas 26 a 31). A estos métodos también se les llama **servicios public** o la **interfaz public** que proporciona la clase a sus clientes.

En este ejemplo, la clase Tiempo1 no declara un constructor, por lo que tiene un constructor predeterminado que le suministra el compilador. Cada variable de instancia recibe en forma implícita el valor predeterminado 0 para un int. Observe que las variables de instancia también pueden inicializarse cuando se declaran en el cuerpo de la clase, usando la misma sintaxis de inicialización que la de una variable local.

El método establecerTiempo (líneas 12 a 17) es un método public que declara tres parámetros int y los utiliza para establecer la hora. Una expresión condicional evalúa cada argumento, para determinar si el valor se encuentra en un rango especificado. Por ejemplo, el valor de hora (línea 14) debe ser mayor o igual que 0 y menor que 24, ya que el formato de hora universal representa las horas como enteros de 0 a 23 (por ejemplo, la 1 PM es la hora 13 y las 11 PM son la hora 23; medianoche es la hora 0 y mediodía es la hora 12). De manera similar,

```
1  // Fig. 8.1: Tiempo1.java
2  // La declaración de la clase Tiempo1 mantiene la hora en formato de 24 horas.
3
4  public class Tiempo1
5  {
6     private int hora;   // 0 - 23
7     private int minuto; // 0 - 59
8     private int segundo; // 0 - 59
9
10    // establece un nuevo valor de tiempo, usando la hora universal; asegura que
11    // los datos sean consistentes, al establecer los valores inválidos a cero
12    public void establecerTiempo( int h, int m, int s )
13    {
14       hora = ( ( h >= 0 && h < 24 ) ? h : 0 );    // valida la hora
15       minuto = ( ( m >= 0 && m < 60 ) ? m : 0 );  // valida el minuto
16       segundo = ( ( s >= 0 && s < 60 ) ? s : 0 ); // valida el segundo
17    } // fin del método establecerTiempo
18
19    // convierte a objeto String en formato de hora universal (HH:MM:SS)
20    public String aStringUniversal()
21    {
22       return String.format( "%02d:%02d:%02d", hora, minuto, segundo );
23    } // fin del método aStringUniversal
24
25    // convierte a objeto String en formato de hora estándar (H:MM:SS AM o PM)
26    public String toString()
27    {
28       return String.format( "%d:%02d:%02d %s",
29          ( ( hora == 0 || hora == 12 ) ? 12 : hora % 12 ),
30          minuto, segundo, ( hora < 12 ? "AM" : "PM" ) );
31    } // fin del método toString
32 } // fin de la clase Tiempo1
```

Figura 8.1 | La declaración de la clase Tiempo1 mantiene la hora en formato de 24 horas.

los valores de `minuto` y `segundo` (líneas 15 y 16) deben ser mayores o iguales que 0 y menores que 60. Cualquier valor fuera de estos rangos se establece como cero para asegurar que un objeto `Tiempo1` siempre contenga *datos consistentes*; esto es, los valores de datos del objeto siempre se mantienen en rango, aun si los valores que se proporcionan como argumentos para el método `establecerTiempo` son *incorrectos*. En este ejemplo, cero es un valor consistente para `hora`, `minuto` y `segundo`.

Un valor que se pasa a `establecerTiempo` es correcto si se encuentra dentro del rango permitido para el miembro que va a inicializar. Por lo tanto, cualquier número en el rango de 0 a 23 sería un valor correcto para la `hora`. Un valor correcto siempre es un valor consistente. Sin embargo, un valor consistente no es necesariamente un valor correcto. Si `establecerTiempo` establece `hora` a `0` debido a que el argumento que recibió se encontraba fuera del rango, entonces `establecerTiempo` está recibiendo un valor incorrecto y lo hace consistente, para que el objeto permanezca en un estado consistente en todo momento. La hora *correcta* del día podrían ser las 11 AM, pero debido a que la persona pudo haber introducido en forma accidental una hora fuera de rango (incorrecta), optamos por establecer la hora al valor consistente de cero. En este caso, tal vez sea conveniente indicar que el objeto es incorrecto. En el capítulo 13, Manejo de excepciones, aprenderá técnicas elegantes que permitirán a sus clases indicar cuándo se reciben valores incorrectos.



Observación de ingeniería de software 8.1

Los métodos que modifican los valores de variables `private` deben verificar que los nuevos valores que se les pretende asignar sean apropiados. Si no lo son, deben colocar las variables `private` en un estado consistente apropiado.

El método `aStringUniversal` (líneas 20 a 23) no recibe argumentos y devuelve un objeto `String` en formato de hora universal, el cual consiste de seis dígitos: dos para la hora, dos para los minutos y dos para los segundos. Por ejemplo, si la hora es 1:30:07 PM, el método `aStringUniversal` devuelve `13:30:07`. La instrucción `return` (línea 22) utiliza el método `static format` de la clase `String` para devolver un objeto `String` que contiene los valores con formato de `hora`, `minuto` y `segundo`, cada uno con dos dígitos y posiblemente, un `0` a la izquierda (el cual se especifica con la bandera `0`). El método **format** es similar al método `System.out.printf`, sólo que `format` devuelve un objeto `String` con formato, en vez de mostrarlo en una ventana de comandos. El método `aStringUniversal` devuelve el objeto `String` con formato.

El método `toString` (líneas 26 a 31) no recibe argumentos y devuelve un objeto `String` en formato de hora estándar, el cual consiste en los valores de `hora`, `minuto` y `segundo` separados por signos de dos puntos (:), y seguidos de un indicador AM o PM (por ejemplo, `1:27:06 PM`). Al igual que el método `aStringUniversal`, el método `toString` utiliza el método `static String format` para dar formato a los valores de `minuto` y `segundo` como valores de dos dígitos con 0s a la izquierda, en caso de ser necesario. La línea 29 utiliza un operador condicional (`?:`) para determinar el valor de `hora` en la cadena; si `hora` es `0` o `12` (AM o PM), aparece como 12; en cualquier otro caso, aparece como un valor de 1 a 11. El operador condicional en la línea 30 determina si se devolverá AM o PM como parte del objeto `String`.

En la sección 6.4 vimos que todos los objetos en Java tienen un método `toString` que devuelve una representación `String` del objeto. Optamos por devolver un objeto `String` que contiene la hora en formato estándar. El método `toString` se puede llamar en forma implícita cada vez que aparece un objeto `Tiempo1` en el código, en donde se necesita un `String`, como el valor para imprimir con un especificador de formato **%s** en una llamada a `System.out.printf`.

Uso de la clase `Tiempo1`

Como aprendió en el capítulo 3, cada clase que se declara representa un nuevo tipo en Java. Por lo tanto, después de declarar la clase `Tiempo1`, podemos utilizarla como un tipo en las declaraciones como

```
Tiempo1  puestasol; // puestasol puede guardar una referencia a un objeto Tiempo1
```

La clase de la aplicación `PruebaTiempo1` (figura 8.2) utiliza la clase `Tiempo1`. La línea 9 declara y crea un objeto `Tiempo1` y lo asigna a la variable local `tiempo`. Observe que `new` invoca en forma implícita al constructor predeterminado de la clase `Tiempo1`, ya que `Tiempo1` no declara constructores. Las líneas 12 a 16 imprimen en pantalla la hora, primero en formato universal (mediante la invocación al método `aStringUniversal` en la línea 13) y después en formato estándar (mediante la invocación explícita del método `toString` de `tiempo` en la línea 15) para confirmar que el objeto `Tiempo1` se haya inicializado en forma apropiada.

```
1  // Fig. 8.2: PruebaTiempo1.java
2  // Objeto Tiempo1 utilizado en una aplicación.
3
4  public class PruebaTiempo1
5  {
6     public static void main( String args[] )
7     {
8        // crea e inicializa un objeto Tiempo1
9        Tiempo1 tiempo = new Tiempo1(); // invoca el constructor de Tiempo1
10
11       // imprime representaciones de cadena del tiempo
12       System.out.print( "La hora universal inicial es: " );
13       System.out.println( tiempo.aStringUniversal() );
14       System.out.print( "La hora estandar inicial es: " );
15       System.out.println( tiempo.toString() );
16       System.out.println(); // imprime una línea en blanco
17
18       // modifica el tiempo e imprime el tiempo actualizado
19       tiempo.establecerTiempo( 13, 27, 6 );
20       System.out.print( "La hora universal despues de establecerTiempo es: " );
21       System.out.println( tiempo.aStringUniversal() );
22       System.out.print( "La hora estandar despues de establecerTiempo es: " );
23       System.out.println( tiempo.toString() );
24       System.out.println(); // imprime una línea en blanco
25
26       // establece el tiempo con valores inválidos; imprime el tiempo actualizado
27       tiempo.establecerTiempo( 99, 99, 99 );
28       System.out.println( "Despues de intentar ajustes invalidos:" );
29       System.out.print( "Hora universal: " );
30       System.out.println( tiempo.aStringUniversal() );
31       System.out.print( "Hora estandar: " );
32       System.out.println( tiempo.toString() );
33    } // fin de main
34 } // fin de la clase PruebaTiempo1
```

```
La hora universal inicial es: 00:00:00
La hora estandar inicial es: 12:00:00 AM

La hora universal despues de establecerTiempo es: 13:27:06
La hora estandar despues de establecerTiempo es: 1:27:06 PM

Despues de intentar ajustes invalidos:
Hora universal: 00:00:00
Hora estandar: 12:00:00 AM
```

Figura 8.2 | Objeto `Tiempo1` utilizado en una aplicación.

La línea 19 invoca al método `establecerTiempo` del objeto `tiempo` para modificar la hora. Después las líneas 20 a 24 imprimen en pantalla la hora otra vez en ambos formatos, para confirmar que la hora se haya ajustado en forma apropiada.

Para ilustrar que el método `establecerTiempo` mantiene el objeto en un estado consistente, la línea 27 llama al método `establecerTiempo` con los argumentos de 99 para la `hora`, el `minuto` y el `segundo`. Las líneas 28 a 32 imprimen de nuevo el tiempo en ambos formatos, para confirmar que `establecerTiempo` haya mantenido el estado consistente del objeto, y después el programa termina. Las últimas dos líneas de la salida de la aplicación muestran que el tiempo se restablece a medianoche (el valor inicial de un objeto `Tiempo1`) si tratamos de establecer el tiempo con tres valores fuera de rango.

Notas acerca de la declaración de la clase `Tiempo1`

Es necesario considerar diversas cuestiones sobre el diseño de clases, en relación con la clase `Tiempo1`. Las variables de instancia `hora`, `minuto` y `segundo` se declaran como `private`. La representación de datos que se utilice dentro de la clase no concierne a los clientes de la misma. Por ejemplo, sería perfectamente razonable que `Tiempo1` representara el tiempo internamente como el número de segundos transcurridos a partir de medianoche, o el número de minutos y segundos transcurridos a partir de medianoche. Los clientes podrían usar los mismos métodos `public` para obtener los mismos resultados, sin tener que preocuparse por lo anterior. (El ejercicio 8.5 le pide que represente la hora en la clase `Tiempo1` como el número de segundos transcurridos a partir de medianoche, y que muestre que, en definitiva, no hay cambios visibles para los clientes de la clase).



Observación de ingeniería de software 8.2

Las clases simplifican la programación, ya que el cliente sólo puede utilizar los métodos `public` *expuestos por la clase. Dichos miembros, por lo general, están orientados a los clientes, en vez de estar orientados a la implementación. Los clientes nunca se percatan de (ni se involucran en) la implementación de una clase. Por lo normal se preocupan acerca de lo que hace la clase, pero no cómo lo hace.*



Observación de ingeniería de software 8.3

Las interfaces cambian con menos frecuencia que las implementaciones. Cuando cambia una implementación, el código dependiente de esa implementación debe cambiar de manera acorde. El ocultamiento de la implementación reduce la posibilidad de que otras partes del programa se vuelvan dependientes de los detalles de la implementación de la clase.

8.3 Control del acceso a los miembros

Los modificadores de acceso `public` y `private` controlan el acceso a las variables y los métodos de una clase (en el capítulo 9, presentaremos el modificador de acceso adicional `protected`). Como dijimos en la sección 8.2, el principal propósito de los métodos `public` es presentar a los clientes de la clase una vista de los servicios que proporciona (la interfaz pública de la clase). Los clientes de la clase no necesitan preocuparse por la forma en que la clase realiza sus tareas. Por esta razón, las variables y métodos `private` de una clase (es decir, los detalles de implementación de la clase) no son directamente accesibles para los clientes de la clase.

La figura 8.3 demuestra que los miembros de una clase `private` no son directamente accesibles fuera de la clase. Las líneas 9 a 11 tratan de acceder en forma directa a las variables de instancia `private hora`, `minuto` y `segundo` del objeto `tiempo` de la clase `Tiempo1`. Al compilar este programa, el compilador genera mensajes de error que indican que estos miembros `private` no son accesibles. [*Nota:* este programa asume que se utiliza la clase `Tiempo1` de la figura 8.1].



Error común de programación 8.1

Cuando un método que no es miembro de una clase trata de acceder a un miembro `private` *de esa clase, se produce un error de compilación.*

```
1  // Fig. 8.3: PruebaAccesoMiembros.java
2  // Los miembros private de la clase Tiempo1 no son accesibles.
3  public class PruebaAccesoMiembros
4  {
5     public static void main( String args[] )
6     {
7        Tiempo1 tiempo = new Tiempo1(); // crea e inicializa un objeto Tiempo1
8
9        tiempo.hora = 7;     // error: hora tiene acceso privado en Tiempo1
10       tiempo.minuto = 15;  // error: minuto tiene acceso privado en Tiempo1
11       tiempo.segundo = 30; // error: segundo tiene acceso privado en Tiempo1
12    } // fin de main
13 } // fin de la clase PruebaAccesoMiembros
```

Figura 8.3 | Los miembros private de la clase `Tiempo1` no son accesibles. (Parte 1 de 2).

```
PruebaAccesoMiembros.java:9: hora has private access in Tiempo1
      tiempo.hora = 7;    // error: hora tiene acceso privado en Tiempo1
            ^
PruebaAccesoMiembros.java:10: minuto has private access in Tiempo1
      tiempo.minuto = 15; // error: minuto tiene acceso privado en Tiempo1
            ^
PruebaAccesoMiembros.java:11: segundo has private access in Tiempo1
      tiempo.segundo = 30; // error: segundo tiene acceso privado en Tiempo1
            ^
3 errors
```

Figura 8.3 | Los miembros private de la clase `Tiempo1` no son accesibles. (Parte 2 de 2).

8.4 Referencias a los miembros del objeto actual mediante `this`

Cada objeto puede acceder a una referencia a sí mismo mediante la palabra clave **this** (también conocida como **referencia this**). Cuando se hace una llamada a un método no `static` para un objeto específico, el cuerpo del método utiliza en forma implícita la palabra clave `this` para hacer referencia a las variables de instancia y los demás métodos del objeto. Como verá en la figura 8.4, puede utilizar también la palabra clave `this` explícitamente en el cuerpo de un método no `static`. La sección 8.5 muestra otro uso interesante de la palabra clave `this`. La sección 8.11 explica por qué no puede usarse la palabra clave `this` en un método `static`.

Ahora demostraremos el uso implícito y explícito de la referencia `this` para permitir al método `main` de la clase `PruebaThis` que muestre en pantalla los datos `private` de un objeto de la clase `TiempoSimple` (figura 8.4). Observe que este ejemplo es el primero en el que declaramos dos clases en un archivo; la clase `PruebaThis` se declara en las líneas 4 a 11 y la clase `TiempoSimple` se declara en las líneas 14 a 47. Hicimos esto para demostrar que, al compilar un archivo `.java` que contiene más de una clase, el compilador produce un archivo de clase separado con la extensión `.class` para cada clase compilada. En este caso se produjeron dos archivos separados: `TiempoSimple.class` y `PruebaThis.class`. Cuando un archivo de código fuente (`.java`) contiene varias declaraciones de clases, el compilador coloca los archivos para esas clases en el mismo directorio. Observe además que sólo la clase `PruebaThis` se declara `public` en la figura 8.4. Un archivo de código fuente sólo puede contener una clase `public`; de lo contrario, se produce un error de compilación.

```
1  // Fig. 8.4: PruebaThis.java
2  // Uso implícito y explícito de this para hacer referencia a los miembros de un objeto.
3
4  public class PruebaThis
5  {
6     public static void main( String args[] )
7     {
8        TiempoSimple tiempo = new TiempoSimple( 15, 30, 19 );
9        System.out.println( tiempo.crearString() );
10    } // fin de main
11 } // fin de la clase PruebaThis
12
13 // la clase TiempoSimple demuestra la referencia "this"
14 class TiempoSimple
15 {
16    private int hora;    // 0-23
17    private int minuto;  // 0-59
18    private int segundo; // 0-59
19
20    // si el constructor utiliza nombres de parámetros idénticos a
```

Figura 8.4 | Uso implícito y explícito de `this` para hacer referencia a los miembros de un objeto. (Parte 1 de 2).

```
21      // los nombres de las variables de instancia, se requiere la
22      // referencia “this” para diferenciar unos nombres de otros
23      public TiempoSimple( int hora, int minuto, int segundo )
24      {
25         this.hora = hora;        // establece la hora del objeto "this"
26         this.minuto = minuto;    // establece el minuto del objeto "this"
27         this.segundo = segundo;  // establece el segundo del objeto "this"
28      } // fin del constructor de TiempoSimple
29
30      // usa la referencia "this" explícita e implícita para llamar aStringUniversal
31      public String crearString()
32      {
33         return String.format( "%24s: %s\n%24s: %s",
34            "this.aStringUniversal()", this.aStringUniversal(),
35            "aStringUniversal()", aStringUniversal() );
36      } // fin del método crearString
37
38      // convierte a String en formato de hora universal (HH:MM:SS)
39      public String aStringUniversal()
40      {
41         // "this" no se requiere aquí para acceder a las variables de instancia,
42         // ya que el método no tiene variables locales con los mismos
43         // nombres que las variables de instancia
44         return String.format( "%02d:%02d:%02d",
45            this.hora, this.minuto, this.segundo );
46      } // fin del método aStringUniversal
47   } // fin de la clase TiempoSimple
```

```
this.aStringUniversal(): 15:30:19
     aStringUniversal(): 15:30:19
```

Figura 8.4 | Uso implícito y explícito de `this` para hacer referencia a los miembros de un objeto. (Parte 2 de 2).

La clase `TiempoSimple` (líneas 14 a 47) declara tres variables de instancia `private`: `hora`, `minuto` y `segundo` (líneas 16 a 18). El constructor (líneas 23 a 28) recibe tres argumentos `int` para inicializar un objeto `TiempoSimple`. Observe que para el constructor (línea 23) utilizamos nombres de parámetros idénticos a los nombres de las variables de instancia de la clase (líneas 16 a 18). No recomendamos esta práctica, pero lo hicimos aquí para ocultar las variables de instancia correspondientes y así poder ilustrar el uso explícito de la referencia `this`. Si un método contiene una variable local con el mismo nombre que el de un campo, hará referencia a la variable local en vez del campo. En este caso, la variable local oculta el campo en el alcance del método. No obstante, el método puede utilizar la referencia `this` para hacer referencia al campo oculto de manera explícita, como se muestra en las líneas 25 a 27 para las variables de instancia ocultas de `TiempoSimple`.

El método `crearString` (líneas 31 a 36) devuelve un objeto `String` creado por una instrucción que utiliza la referencia `this` en forma explícita e implícita. La línea 34 utiliza la referencia `this` en forma explícita para llamar al método `aStringUniversal`. La línea 35 usa la referencia `this` en forma implícita para llamar al mismo método. Observe que ambas líneas realizan la misma tarea. Por lo general, los programadores no utilizan la referencia `this` en forma explícita para hacer referencia a otros métodos en el objeto actual. Además, observe que la línea 45 en el método `aStringUniversal` utiliza en forma explícita la referencia `this` para acceder a cada variable de instancia. Esto no es necesario aquí, ya que el método no tiene variables locales que oculten las variables de instancia de la clase.



Error común de programación 8.2

A menudo se produce un error lógico cuando un método contiene un parámetro o variable local con el mismo nombre que un campo de la clase. En tal caso, use la referencia `this` *si desea acceder al campo de la clase; de no ser así, se hará referencia al parámetro o variable local del método.*



Tip para prevenir errores 8.1

Evite los nombres de los parámetros o variables locales que tengan conflicto con los nombres de los campos. Esto ayuda a evitar errores sutiles, difíciles de localizar.

La clase de la aplicación `PruebaThis` (líneas 4 a 11) demuestra el uso de la clase `TiempoSimple`. La línea 8 crea una instancia de la clase `TiempoSimple` e invoca a su constructor. La línea 9 invoca al método `crearString` del objeto y después muestra los resultados en pantalla.



Tip de rendimiento 8.1

Para conservar la memoria, Java mantiene sólo una copia de cada método por clase; todos los objetos de la clase invocan a este método. Por otro lado, cada objeto tiene su propia copia de las variables de instancia de la clase (es decir, las variables no `static`). Cada método de la clase utiliza en forma implícita la referencia `this` para determinar el objeto específico de la clase que se manipulará.

8.5 Ejemplo práctico de la clase `Tiempo`: constructores sobrecargados

Como sabe, puede declarar su propio constructor para especificar cómo deben inicializarse los objetos de una clase. A continuación demostraremos una clase con varios **constructores sobrecargados**, que permiten a los objetos de esa clase inicializarse de distintas formas. Para sobrecargar los constructores, sólo hay que proporcionar varias declaraciones del constructor con distintas firmas. En la sección 6.12 vimos que el compilador diferencia las firmas en base al número, tipos y orden de los parámetros en cada firma.

La clase `Tiempo2` con constructores sobrecargados

El constructor predeterminado de la clase `Tiempo1` (figura 8.1) inicializó `hora`, `minuto` y `segundo` con sus valores predeterminados de `0` (medianoche en formato de hora universal). El constructor predeterminado no permite que los clientes de la clase inicialicen la hora con valores específicos distintos de cero. La clase `Tiempo2` (figura 8.5) contiene cinco constructores sobrecargados que proporcionan formas convenientes para inicializar los objetos de la nueva clase `Tiempo2`. Cada constructor inicializa el objeto para que empiece en un estado consistente. En este programa, cuatro de los constructores invocan un quinto constructor, el cual a su vez llama al método `establecerTiempo` para asegurar que el valor suministrado para `hora` se encuentre en el rango de 0 a 23, y que los valores para `minuto` y `segundo` se encuentren cada uno en el rango de 0 a 59. Si un valor está fuera de rango, se establece a 0 mediante `establecerTiempo` (una vez más se asegura que cada variable de instancia permanezca en un estado consistente). Para invocar el constructor apropiado, el compilador relaciona el número, los tipos y el orden de los argumentos especificados en la llamada al constructor con el número, los tipos y el orden de los tipos de los parámetros especificados en la declaración de cada constructor. Observe que la clase `Tiempo2` también proporciona métodos *establecer* y *obtener* para cada variable de instancia.

```
// Fig. 8.5: Tiempo2.java
// Declaración de la clase Tiempo2 con constructores sobrecargados.

public class Tiempo2
{
   private int hora;    // 0 - 23
   private int minuto;  // 0 - 59
   private int segundo; // 0 - 59

   // Constructor de Tiempo2 sin argumentos: inicializa cada variable de instancia
   // a cero; asegura que los objetos Tiempo2 empiecen en un estado consistente
   public Tiempo2()
   {
```

Figura 8.5 | La clase `Tiempo2` con constructores sobrecargados. (Parte 1 de 3).

```
      this( 0, 0, 0 ); // invoca al constructor de Tiempo2 con tres argumentos
   } // fin del constructor de Tiempo2 sin argumentos

   // Constructor de Tiempo2: se suministra hora, minuto y segundo con valor predeterminado de 0
   public Tiempo2( int h )
   {
      this( h, 0, 0 ); // invoca al constructor de Tiempo2 con tres argumentos
   } // fin del constructor de Tiempo2 con un argumento

   // Constructor de Tiempo2: se suministran hora y minuto, segundo con valor predeterminado de 0
   public Tiempo2( int h, int m )
   {
      this( h, m, 0 ); // invoca al constructor de Tiempo2 con tres argumentos
   } // fin del constructor de Tiempo2 con dos argumentos

   // Constructor de Tiempo2: se suministran hora, minuto y segundo
   public Tiempo2( int h, int m, int s )
   {
      establecerTiempo( h, m, s ); // invoca a establecerTiempo para validar el tiempo
   } // fin del constructor de Tiempo2 con tres argumentos

   // Constructor de Tiempo2: se suministra otro objeto Tiempo2
   public Tiempo2( Tiempo2 tiempo )
   {
      // invoca al constructor de Tiempo2 con tres argumentos
      this( tiempo.obtenerHora(), tiempo.obtenerMinuto(), tiempo.obtenerSegundo() );
   } // fin del constructor de Tiempo2 con un objeto Tiempo2 como argumento

   // Métodos "establecer"
   // establece un nuevo valor de tiempo usando la hora universal; asegura que
   // los datos sean consistentes, estableciendo los valores inválidos en cero
   public void establecerTiempo( int h, int m, int s )
   {
      establecerHora( h );    // establece la hora
      establecerMinuto( m );  // establece el minuto
      establecerSegundo( s ); // establece el segundo
   } // fin del método establecerTiempo

   // valida y establece la hora
   public void establecerHora( int h )
   {
      hora = ( ( h >= 0 && h < 24 ) ? h : 0 );
   } // fin del método establecerHora

   // valida y establece el minuto
   public void establecerMinuto( int m )
   {
      minuto = ( ( m >= 0 && m < 60 ) ? m : 0 );
   } // fin del método establecerMinuto

   // valida y establece el segundo
   public void establecerSegundo( int s )
   {
      segundo = ( ( s >= 0 && s < 60 ) ? s : 0 );
   } // fin del método establecerSegundo

   // Métodos "obtener"
```

Figura 8.5 | La clase `Tiempo2` con constructores sobrecargados. (Parte 2 de 3).

```
71      // obtiene el valor de la hora
72      public int obtenerHora()
73      {
74         return hora;
75      } // fin del método obtenerHora
76
77      // obtiene el valor del minuto
78      public int obtenerMinuto()
79      {
80         return minuto;
81      } // fin del método obtenerMinuto
82
83      // obtiene el valor del segundo
84      public int obtenerSegundo()
85      {
86         return segundo;
87      } // fin del método obtenerSegundo
88
89      // convierte a String en formato de hora universal (HH:MM:SS)
90      public String aStringUniversal()
91      {
92         return String.format(
93            "%02d:%02d:%02d", obtenerHora(), obtenerMinuto(), obtenerSegundo() );
94      } // fin del método aStringUniversal
95
96      // convierte a String en formato de hora estándar (H:MM:SS AM o PM)
97      public String toString()
98      {
99         return String.format( "%d:%02d:%02d %s",
100           ( (obtenerHora() == 0 || obtenerHora() == 12) ? 12 : obtenerHora() % 12 ),
101           obtenerMinuto(), obtenerSegundo(), ( obtenerHora() < 12 ? "AM" : "PM" ) );
102     } // fin del método toString
103  } // fin de la clase Tiempo2
```

Figura 8.5 | La clase `Tiempo2` con constructores sobrecargados. (Parte 3 de 3).

Constructores de la clase `Tiempo2`

Las líneas 12 a 15 declaran un **constructor sin argumentos**; es decir, un constructor que se invoca sin argumentos; simplemente inicializa el objeto como se especifica en el cuerpo del constructor. En el cuerpo, presentamos un uso de la referencia `this` que se permite sólo como la primera instrucción en el cuerpo de un constructor. La línea 14 utiliza a `this` en la sintaxis de la llamada al método para invocar al constructor de `Tiempo2` que recibe tres argumentos (líneas 30 a 33). El constructor sin argumentos pasa los valores de `0` para `hora`, `minuto` y `segundo` al constructor con tres parámetros. El uso de la referencia `this` que se muestra aquí es una forma popular de reutilizar el código de inicialización que proporciona otro de los constructores de la clase, en vez de definir código similar en el cuerpo del constructor sin argumentos. Utilizamos esta sintaxis en cuatro de los cinco constructores de `Tiempo2` para que la clase sea más fácil de mantener y modificar. Si necesitamos cambiar la forma en que se inicializan los objetos de la clase `Tiempo2`, sólo hay que modificar el constructor al que necesitan llamar los demás constructores de la clase. Incluso hasta ese constructor podría no requerir de modificación en este ejemplo. Simplemente llama al método `establecerTiempo` para realizar la verdadera inicialización, por lo que es posible que los cambios que pudiera requerir la clase se localicen en los métodos *establecer*.



Error común de programación 8.3

Es un error de sintaxis utilizar `this` *en el cuerpo de un constructor para llamar a otro constructor de la misma clase, si esa llamada no es la primera instrucción en el constructor. También es un error de sintaxis cuando un método trata de invocar a un constructor directamente, mediante* `this`*.*

Las líneas 18 a 21 declaran un constructor de `Tiempo2` con un solo parámetro `int` que representa la `hora`, que se pasa con 0 para `minuto` y `segundo` al constructor de las líneas 30 a 33. Las líneas 24 a 27 declaran un constructor de `Tiempo2` que recibe dos parámetros `int`, los cuales representan la `hora` y el `minuto`, que se pasan con un `0` para `segundo` al constructor de las líneas 30 a 33. Al igual que el constructor sin argumentos, cada uno de estos constructores invoca al constructor en las líneas 30 a 33 para minimizar la duplicación de código. Las líneas 30 a 33 declaran el constructor `Tiempo2` que recibe tres parámetros `int`, los cuales representan la `hora`, el `minuto` y el `segundo`. Este constructor llama a `establecerTiempo` para inicializar las variables de instancia con valores consistentes.



Error común de programación 8.4

Un constructor puede llamar a los métodos de la clase. Tenga en cuenta que tal vez las variables de instancia no se encuentren aún en un estado consistente, ya que el constructor está en el proceso de inicializar el objeto. El uso de variables de instancia antes de inicializarlas en forma apropiada es un error lógico.

Las líneas 36 a 40 declaran un constructor de `Tiempo2` que recibe una referencia `Tiempo2` a otro objeto `Tiempo2`. En este caso, los valores del argumento `Tiempo2` se pasan al constructor de tres argumentos en las líneas 30 a 33 para inicializar `hora`, `minuto` y `segundo`. Observe que la línea 39 podría haber accedido en forma directa a los valores `hora`, `minuto` y `segundo` del argumento `tiempo` del constructor con las variables de instancia `tiempo.hora`, `tiempo.minuto` y `tiempo.segundo`, aun cuando `hora`, `minuto` y `segundo` se declaran como variables `private` de la clase `Tiempo2`. Esto se debe a una relación especial entre los objetos de la misma clase. En un momento veremos por qué es preferible utilizar los métodos *obtener*.



Observación de ingeniería de software 8.4

Cuando un objeto de una clase tiene una referencia a otro objeto de la misma clase, el primer objeto puede acceder a todos los datos y métodos del segundo objeto (incluyendo los que sean `private`*).*

Observaciones en relación con los métodos `Establecer` y `Obtener`, y los constructores de la clase `Tiempo2`

Observe que los métodos *establecer* y *obtener* de `Tiempo2` se llaman en el cuerpo de la clase. En especial, el método `establecerTiempo` llama a los métodos `establecerHora`, `establecerMinuto` y `establecerSegundo` en las líneas 47 a 49, y los métodos `aStringUniversal` y `toString` llaman a los métodos `obtenerHora`, `obtenerMinuto` y `obtenerSegundo` en la línea 93 y en las líneas 100 y 101, respectivamente. En cada caso, estos métodos podrían haber accedido a los datos `private` de la clase en forma directa, sin necesidad de llamar a los métodos *establecer* y *obtener*. Sin embargo, considere la acción de cambiar la representación del tiempo, de tres valores `int` (que requieren 12 bytes de memoria) a un solo valor `int` que represente el número total de segundos transcurridos a partir de medianoche (que requiere sólo 4 bytes de memoria). Si hacemos ese cambio, sólo tendrían que cambiar los cuerpos de los métodos que acceden directamente a los datos `private`; en especial, los métodos *establecer* y *obtener* individuales para `hora`, `minuto` y `segundo`. No habría necesidad de modificar los cuerpos de los métodos `establecerTiempo`, `aStringUniversal` o `toString`, ya que no acceden directamente a los datos. Si se diseña la clase de esta forma, se reduce la probabilidad de que se produzcan errores de programación al momento de alterar la implementación de la clase.

De manera similar, cada constructor de `Tiempo2` podría escribirse de forma que incluya una copia de las instrucciones apropiadas de los métodos `establecerHora`, `establecerMinuto` y `establecerSegundo`. Esto sería un poco más eficiente, ya que se eliminan la llamada extra al constructor y la llamada a `establecerTiempo`. No obstante, duplicar las instrucciones en varios métodos o constructores dificulta más el proceso de modificar la representación de datos interna de la clase. Si hacemos que los constructores de `Tiempo2` llamen al constructor con tres argumentos (o que incluso llamen a `establecerTiempo` directamente), cualquier modificación a la implementación de `establecerTiempo` sólo tendrá que hacerse una vez.



Observación de ingeniería de software 8.5

Al implementar un método de una clase, use los métodos establecer y obtener de la clase para acceder a sus datos `private`*. Esto simplifica el mantenimiento del código y reduce la probabilidad de errores.*

Uso de los constructores sobrecargados de la clase **Tiempo2**

La clase PruebaTiempo2 (figura 8.6) crea seis objetos Tiempo2 (líneas 8 a 13) para invocar a los constructores sobrecargados de Tiempo2. La línea 8 muestra que para invocar el constructor sin argumentos (líneas 12 a 15 de la figura 8.5) se coloca un conjunto vacío de paréntesis después del nombre de la clase, cuando se asigna un objeto Tiempo2 mediante new. Las líneas 9 a 13 del programa demuestran el paso de argumentos a los demás constructores de Tiempo2. La línea 9 invoca al constructor en las líneas 18 a 21 de la figura 8.5. La línea 10 invoca al constructor en las líneas 24 a 27 de la figura 8.5. Las líneas 11 y 12 invocan al constructor en las líneas 30 a 33 de la figura 8.5. La línea 13 invoca al constructor en las líneas 36 a 40 de la figura 8.5. La aplicación muestra en pantalla la representación String de cada objeto Tiempo2 inicializado, para confirmar que cada uno de ellos se haya inicializado en forma apropiada.

```
1  // Fig. 8.6: PruebaTiempo2.java
2  // Uso de constructores sobrecargados para inicializar objetos Tiempo2.
3
4  public class PruebaTiempo2
5  {
6     public static void main( String args[] )
7     {
8        Tiempo2 t1 = new Tiempo2();             // 00:00:00
9        Tiempo2 t2 = new Tiempo2( 2 );          // 02:00:00
10       Tiempo2 t3 = new Tiempo2( 21, 34 );     // 21:34:00
11       Tiempo2 t4 = new Tiempo2( 12, 25, 42 ); // 12:25:42
12       Tiempo2 t5 = new Tiempo2( 27, 74, 99 ); // 00:00:00
13       Tiempo2 t6 = new Tiempo2( t4 );         // 12:25:42
14
15       System.out.println( "Se construyo con:" );
16       System.out.println( "t1: todos los argumentos predeterminados" );
17       System.out.printf( "   %s\n", t1.aStringUniversal() );
18       System.out.printf( "   %s\n", t1.toString() );
19       System.out.println(
20          "t2: se especifico hora; minuto y segundo predeterminados" );
21       System.out.printf( "   %s\n", t2.aStringUniversal() );
22       System.out.printf( "   %s\n", t2.toString() );
23
24       System.out.println(
25          "t3: se especificaron hora y minuto; segundo predeterminado" );
26       System.out.printf( "   %s\n", t3.aStringUniversal() );
27       System.out.printf( "   %s\n", t3.toString() );
28
29       System.out.println( "t4: se especificaron hora, minuto y segundo" );
30       System.out.printf( "   %s\n", t4.aStringUniversal() );
31       System.out.printf( "   %s\n", t4.toString() );
32
33       System.out.println( "t5: se especificaron todos los valores invalidos" );
34       System.out.printf( "   %s\n", t5.aStringUniversal() );
35       System.out.printf( "   %s\n", t5.toString() );
36
37       System.out.println( "t6: se especifico el objeto t4 de Tiempo2" );
38       System.out.printf( "   %s\n", t6.aStringUniversal() );
39       System.out.printf( "   %s\n", t6.toString() );
40    } // fin de main
41 } // fin de la clase PruebaTiempo2
```

```
Se construyo con:
t1: todos los argumentos predeterminados
   00:00:00
   12:00:00 AM
```

Figura 8.6 | Uso de constructores sobrecargados para inicializar objetos Tiempo2. (Parte 1 de 2).

```
t2: se especifico hora; minuto y segundo predeterminados
   02:00:00
   2:00:00 AM
t3: se especificaron hora y minuto; segundo predeterminado
   21:34:00
   9:34:00 PM
t4: se especificaron hora, minuto y segundo
   12:25:42
   12:25:42 PM
t5: se especificaron todos los valores invalidos
   00:00:00
   2:00:00 AM
t6: se especifico el objeto t4 de Tiempo2
   12:25:42
   12:25:42 PM
```

Figura 8.6 | Uso de constructores sobrecargados para inicializar objetos `Tiempo2`. (Parte 2 de 2).

8.6 Constructores predeterminados y sin argumentos

Toda clase debe tener cuando menos un constructor. En la sección 3.7 vimos que si no se proporcionan constructores en la declaración de una clase, el compilador crea un constructor predeterminado que no recibe argumentos cuando se le invoca. El constructor predeterminado inicializa las variables de instancia con los valores iniciales especificados en sus declaraciones, o con sus valores predeterminados (cero para los tipos primitivos numéricos, `false` para los valores `boolean` y `null` para las referencias). En la sección 9.4.1 aprenderá que el constructor predeterminado realiza otra tarea, además de inicializar cada variable de instancia con su valor predeterminado.

Si su clase declara constructores, el compilador no creará un constructor predeterminado. En este caso, para especificar la inicialización predeterminada para objetos de su clase, debe declarar un constructor sin argumentos (como en las líneas 12 a 15 de la figura 8.5). Al igual que un constructor predeterminado, un constructor sin argumentos se invoca con paréntesis vacíos. Observe que el constructor sin argumentos de `Tiempo2` inicializa en forma explícita un objeto `Tiempo2`; para ello pasa un 0 a cada parámetro del constructor con tres argumentos. Como 0 es el valor predeterminado para las variables de instancia `int`, el constructor sin argumentos en este ejemplo podría declararse con un cuerpo vacío. En este caso, cada variable de instancia recibiría su valor predeterminado al momento de llamar al constructor sin argumentos. Si omitimos el constructor sin argumentos, los clientes de esta clase no podrían crear un objeto `Tiempo2` con la expresión `new Tiempo2()`.



Error común de programación 8.5

Si una clase tiene constructores, pero ninguno de los constructores `public` *son sin argumentos, y si un programa intenta llamar a un constructor sin argumentos para inicializar un objeto de esa clase, se produce un error de compilación. Se puede llamar a un constructor sin argumentos sólo cuando la clase no tiene constructores (en cuyo caso se llama al constructor predeterminado), o si la clase tiene un constructor* `public` *sin argumentos.*



Observación de ingeniería de software 8.6

Java permite que otros métodos de la clase, además de sus constructores, tengan el mismo nombre de la clase y especifiquen tipos de valores de retorno. Dichos métodos no son constructores, por lo que no se llaman cuando se crea una instancia de un objeto de la clase. Para determinar cuáles métodos son constructores, Java localiza los métodos que tienen el mismo nombre que la clase y que no especifican un tipo de valor de retorno.

8.7 Observaciones acerca de los métodos *Establecer* y *Obtener*

Como sabe, los campos `private` de una clase pueden manipularse solamente mediante métodos de esa clase. Una manipulación típica podría ser el ajuste del saldo bancario de un cliente (por ejemplo, una variable de instancia `private` de una clase llamada `CuentaBancaria`) mediante un método llamado `calcularInteres`. Las clases a menudo proporcionan métodos `public` para permitir a los clientes de la clase *establecer* (es decir, asignar valores a) u *obtener* (es decir, recibir los valores de) variables de instancia `private`.

Como ejemplo de nomenclatura, un método para establecer la variable de instancia `tasaInteres` se llamaría típicamente `establecerTasaInteres`, y un método para obtener la `tasaDeInteres` se llamaría típicamente `obtenerTasaInteres`. Los métodos *establecer* también se conocen comúnmente como **métodos mutadores**, porque generalmente cambian un valor. Los métodos *obtener* también se conocen comúnmente como **métodos de acceso** o **métodos de consulta**.

Comparación entre los métodos Establecer *y* Obtener*, y los datos* `public`

Parece ser que proporcionar herramientas para *establecer* y *obtener* es esencialmente lo mismo que hacer las variables de instancia `public`. Ésta es una sutileza de Java que hace del lenguaje algo tan deseable para la ingeniería de software. Si una variable de instancia se declara como `public`, cualquier método que tenga una referencia a un objeto que contenga esta variable de instancia podrá leer o escribir en ella. Si una variable de instancia se declara como `private`, un método *obtener* `public` evidentemente permite a otros métodos el acceso a la variable, pero el método *obtener* puede controlar la manera en que el cliente puede tener acceso a la variable. Por ejemplo, un método *obtener* podría controlar el formato de los datos que devuelve y, por ende, proteger el código cliente de la representación actual de los datos. Un método *establecer* `public` puede (y debe) escudriñar cuidadosamente los intentos por modificar el valor de la variable, para asegurar que el nuevo valor sea apropiado para ese elemento de datos. Por ejemplo, un intento por *establecer* el día del mes en una fecha 37 sería rechazado, un intento por *establecer* el peso de una persona en un valor negativo sería rechazado, y así sucesivamente. Entonces, aunque los métodos *establecer* y *obtener* proporcionan acceso a los datos privados, el programador restringe su acceso mediante la implementación de los métodos. Esto ayuda a promover la buena ingeniería de software.

Comprobación de validez en los métodos Establecer

Los beneficios de la integridad de los datos no se dan automáticamente sólo porque las variables de instancia se declaren como `private`; el programador debe proporcionar la comprobación de su validez. Java permite a los programadores diseñar mejores programas de una manera conveniente. Los métodos *establecer* de una clase pueden devolver valores que indiquen que hubo intentos de asignar datos inválidos a los objetos de la clase. Un cliente de la clase puede probar el valor de retorno de un método *establecer* para determinar si el intento del cliente por modificar el objeto tuvo éxito, y entonces tomar la acción apropiada.



Observación de ingeniería de software 8.7

Cuando sea necesario, proporcione métodos `public` *para cambiar y obtener los valores de las variables de instancia* `private`*. Esta arquitectura ayuda a ocultar la implementación de una clase a sus clientes, lo cual mejora la capacidad de modificación de un programa.*



Observación de ingeniería de software 8.8

Los diseñadores de clases no necesitan proporcionar métodos establecer u obtener para cada campo `private`*. Estas capacidades deben proporcionarse solamente cuando esto tenga sentido.*

Métodos predicados

Otro uso común de los métodos de acceso es para evaluar si una condición es verdadera o falsa; por lo general, a dichos métodos se les llama **métodos predicados**. Un ejemplo sería un método `estaVacio` para una **clase contenedora**: una clase capaz de contener muchos objetos, como una lista enlazada, una pila o una cola. (En los capítulos 17 y 19 hablaremos con detalle sobre estas estructuras de datos). Un programa podría evaluar el método `estaVacio` antes de tratar de leer otro elemento de un objeto contenedor. Un programa podría evaluar un método `estaLleno` antes de tratar de insertar otro elemento en un objeto contenedor.

Uso de métodos Establecer *y* Obtener *para crear una clase que sea más fácil de depurar y mantener*

Si sólo un método realiza una tarea específica, como establecer la hora en un objeto `Tiempo2`, es más fácil depurar y mantener esa clase. Si la `hora` no se establece en forma apropiada, el código que modifica la variable de instancia `hora` se localiza en el cuerpo de un método: `establecerHora`. Así, sus esfuerzos de depuración pueden enfocarse en el método `establecerHora`.

8.8 Composición

Una clase puede tener referencias a objetos de otras clases como miembros. A dicha capacidad se le conoce como **composición** y algunas veces como **relación** "***tiene un***". Por ejemplo, un objeto de la clase `RelojAlarma` necesita saber la hora actual y la hora en la que se supone sonará su alarma, por lo que es razonable incluir dos referencias a objetos `Tiempo` como miembros del objeto `RelojAlarma`.



Observación de ingeniería de software 8.9

La composición es una forma de reutilización de software, en donde una clase tiene como miembros referencias a objetos de otras clases.

Nuestro ejemplo de composición contiene tres clases: `Fecha` (figura 8.7), `Empleado` (figura 8.8) y `PruebaEmpleado` (figura 8.9). La clase `Fecha` (figura 8.7) declara las variables de instancia `mes`, `dia` y `anio` (líneas 6 a 8) para representar una fecha. El constructor recibe tres parámetros `int`. La línea 14 invoca el método utilitario `comprobarMes` (líneas 23 a 33) para validar el mes; un valor fuera de rango se establece en `1` para mantener un estado consistente. La línea 15 asume que el valor de `anio` es correcto y no lo valida. La línea 16 invoca al método utilitario `comprobarDia` (líneas 36 a 52) para validar el valor de `dia` con base en el `mes` y `anio` actuales. Las líneas 42 y 43 determinan si el día es correcto, con base en el número de días en el `mes` específico. Si el día no es correcto, las líneas 46 y 47 determinan si el `mes` es Febrero, el día `29` y el `anio` un año bisiesto. Si las líneas 42 a 48 no devuelven un valor correcto para `dia`, la línea 51 devuelve `1` para mantener la `Fecha` en un estado consistente. Observe que las líneas 18 y 19 en el constructor muestran en pantalla la referencia `this` como un objeto `String`. Como `this` es una referencia al objeto `Fecha` actual, se hace una llamada implícita al método `toString` (líneas 55 a 58) para obtener la representación `String` del objeto.

```
1  // Fig. 8.7: Fecha.java
2  // Declaración de la clase Fecha.
3
4  public class Fecha
5  {
6     private int mes;   // 1-12
7     private int dia;   // 1-31 con base en el mes
8     private int anio;  // cualquier año
9
10    // constructor: llama a comprobarMes para confirmar el valor apropiado para el mes;
11    // llama a comprobarDia para confirmar el valor apropiado para el día
12    public Fecha( int elMes, int elDia, int elAnio )
13    {
14       mes = comprobarMes( elMes ); // valida el mes
15       anio = elAnio; // pudo validar el año
16       dia = comprobarDia( elDia ); // valida el día
17
18       System.out.printf(
19          "Constructor de objeto Fecha para la fecha %s\n", this );
20    } // fin del constructor de Fecha
21
22    // método utilitario para confirmar el valor apropiado del mes
23    private int comprobarMes( int mesPrueba )
24    {
25       if ( mesPrueba > 0 && mesPrueba <= 12 ) // valida el mes
26          return mesPrueba;
27       else // mes es inválido
28       {
29          System.out.printf(
30             "Mes invalido (%d) se establecio en 1.", mesPrueba );
31          return 1; // mantiene el objeto en estado consistente
```

Figura 8.7 | Declaración de la clase `Fecha`. (Parte 1 de 2).

```
      } // fin de else
   } // fin del método comprobarMes

   // método utilitario para confirmar el valor apropiado del día, con base en el mes y el año
   private int comprobarDia( int diaPrueba )
   {
      int diasPorMes[] =
         { 0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };

      // comprueba si el día está dentro del rango para el mes
      if ( diaPrueba > 0 && diaPrueba <= diasPorMes[ mes ] )
         return diaPrueba;

      // comprueba si es año bisiesto
      if ( mes == 2 && diaPrueba == 29 && ( anio % 400 == 0 ||
           ( anio % 4 == 0 && anio % 100 != 0 ) ) )
         return diaPrueba;

      System.out.printf( "Dia invalido (%d) se establecio en 1.", diaPrueba );
      return 1;  // mantiene el objeto en estado consistente
   } // fin del método comprobarDia

   // devuelve un objeto String de la forma mes/dia/anio
   public String toString()
   {
      return String.format( "%d/%d/%d", mes, dia, anio );
   } // fin del método toString
} // fin de la clase Fecha
```

Figura 8.7 | Declaración de la clase `Fecha`. (Parte 2 de 2).

La clase `Empleado` (figura 8.8) tiene las variables de instancia `primerNombre`, `apellidoPaterno`, `fechaNacimiento` y `fechaContratacion`. Los miembros `fechaNacimiento` y `fechaContratacion` (líneas 8 y 9) son referencias a objetos `Fecha`. Esto demuestra que una clase puede tener como variables de instancia referencias a objetos de otras clases. El constructor de `Empleado` (líneas 12 a 19) recibe cuatro parámetros: `nombre`, `apellido`, `fechaDeNacimiento` y `fechaDeContratacion`. Los objetos referenciados por los parámetros `fechaDeNacimiento` y `fechaDeContratacion` se asignan a las variables de instancia `fechaNacimiento` y `fechaContratacion` del objeto `Empleado`, respectivamente. Observe que cuando se hace una llamada al método `toString` de la clase `Empleado`, éste devuelve un objeto `String` que contiene las representaciones `String` de los dos objetos `Fecha`. Cada uno de estos objetos `String` se obtiene mediante una llamada implícita al método `toString` de la clase `Fecha`.

```
1  // Fig. 8.8: Empleado.java
2  // Clase Empleado con referencias a otros objetos.
3
4  public class Empleado
5  {
6     private String primerNombre;
7     private String apellidoPaterno;
8     private Fecha fechaNacimiento;
9     private Fecha fechaContratacion;
10
11    // constructor para inicializar nombre, fecha de nacimiento y fecha de contratación
12    public Empleado( String nombre, String apellido, Fecha fechaDeNacimiento,
```

Figura 8.8 | Clase `Empleado` con referencias a otros objetos. (Parte 1 de 2).

```
         Fecha fechaDeContratacion )
      {
         primerNombre = nombre;
         apellidoPaterno = apellido;
         fechaNacimiento = fechaDeNacimiento;
         fechaContratacion = fechaDeContratacion;
      } // fin del constructor de Empleado

      // convierte Empleado a formato String
      public String toString()
      {
         return String.format( "%s, %s  Contratado: %s  Cumpleanios: %s",
            apellidoPaterno, primerNombre, fechaContratacion, fechaNacimiento );
      } // fin del método toString
   } // fin de la clase Empleado
```

Figura 8.8 | Clase `Empleado` con referencias a otros objetos. (Parte 2 de 2).

La clase `PruebaEmpleado` (figura 8.9) crea dos objetos `Fecha` (líneas 8 y 9) para representar la fecha de nacimiento y de contratación de un `Empleado`, respectivamente. La línea 10 crea un `Empleado` e inicializa sus variables de instancia, pasando al constructor dos objetos `String` (que representan el nombre y el apellido del `Empleado`) y dos objetos `Fecha` (que representan la fecha de nacimiento y de contratación). La línea 12 invoca en forma implícita el método `toString` de `Empleado` para mostrar en pantalla los valores de sus variables de instancia y demostrar que el objeto se inicializó en forma apropiada.

```
1   // Fig. 8.9: PruebaEmpleado.java
2   // Demostración de la composición.
3
4   public class PruebaEmpleado
5   {
6      public static void main( String args[] )
7      {
8         Fecha nacimiento = new Fecha( 7, 24, 1949 );
9         Fecha contratacion = new Fecha( 3, 12, 1988 );
10        Empleado empleado = new Empleado( "Bob", "Blue", nacimiento, contratacion );
11
12        System.out.println( empleado );
13     } // fin de main
14  } // fin de la clase PruebaEmpleado
```

```
Constructor de objeto Fecha para la fecha 7/24/1949
Constructor de objeto Fecha para la fecha 3/12/1988
Blue, Bob  Contratado: 3/12/1988  Cumpleanios: 7/24/1949
```

Figura 8.9 | Demostración de la composición.

8.9 Enumeraciones

En la figura 6.9 (`Craps.java`) presentamos el tipo básico `enum`, que define a un conjunto de constantes que se representan como identificadores únicos. En ese programa, las constantes `enum` representaban el estado del juego. En esta sección, hablaremos sobre la relación entre los tipos `enum` y las clases. Al igual que las clases, todos los tipos `enum` son tipos por referencia. Un tipo `enum` se declara con una **declaración enum**, la cual es una lista separada por comas de constantes `enum`; la declaración puede incluir, de manera opcional, otros componentes de las clases tradicionales, como constructores, campos y métodos. Cada declaración `enum` declara a una clase `enum` con las siguientes restricciones:

1. Los tipos `enum` son implícitamente `final`, ya que declaran constantes que no deben modificarse.
2. Las constantes `enum` son implícitamente `static`.
3. Cualquier intento por crear un objeto de un tipo `enum` con el operador `new` produce un error de compilación.

Las constantes `enum` pueden usarse en cualquier parte en donde pueden usarse las constantes, como en las etiquetas `case` de las instrucciones `switch`, y para controlar las instrucciones `for` mejoradas.

La figura 8.10 ilustra cómo declarar variables de instancia, un constructor y varios métodos en un tipo `enum`. La declaración `enum` (líneas 5 a 37) contiene dos partes: las constantes `enum` y los demás miembros del tipo `enum`. La primera parte (líneas 8 a 13) declara seis constantes `enum`. Cada constante `enum` va seguida opcionalmente por argumentos que se pasan al **constructor de `enum`** (líneas 20 a 24). Al igual que los constructores que hemos visto en las clases, un constructor de `enum` puede especificar cualquier número de parámetros, y puede sobrecargarse. En este ejemplo, el constructor de `enum` tiene dos parámetros `String`, por lo que cada constante `enum` va seguida de paréntesis que contienen dos argumentos `String`. La segunda parte (líneas 16 a 36) declara a los demás miembros del tipo `enum`: dos variables de instancia (líneas 16 y 17), un constructor (líneas 20 a 24) y dos métodos (líneas 27 a 30 y líneas 33 a 36).

```
 1  // Fig. 8.10: Libro.java
 2  // Declara un tipo enum con constructor y campos de instancia explícitos,
 3  // junto con métodos de acceso para estos campos
 4
 5  public enum Libro
 6  {
 7     // declara constantes de tipo enum
 8     JHTP6( "Java How to Program 6e", "2005" ),
 9     CHTP4( "C How to Program 4e", "2004" ),
10     IW3HTP3( "Internet & World Wide Web How to Program 3e", "2004" ),
11     CPPHTP4( "C++ How to Program 4e", "2003" ),
12     VBHTP2( "Visual Basic .NET How to Program 2e", "2002" ),
13     CSHARPHTP( "C# How to Program", "2002" );
14
15     // campos de instancia
16     private final String titulo; // título del libro
17     private final String anioCopyright; // año de copyright
18
19     // constructor de enum
20     Libro( String tituloLibro, String anio )
21     {
22        titulo = tituloLibro;
23        anioCopyright = anio;
24     } // fin de constructor de enum Libro
25
26     // método de acceso para el campo titulo
27     public String obtenerTitulo()
28     {
29        return titulo;
30     } // fin del método obtenerTitulo
31
32     // método de acceso para el campo anioCopyright
33     public String obtenerAnioCopyright()
34     {
35        return anioCopyright;
36     } // fin del método obtenerAnioCopyright
37  } // fin de enum Libro
```

Figura 8.10 | Declaración del tipo `enum` con campos de instancia, constructor y métodos.

Las líneas 16 y 17 declaran las variables de instancia `titulo` y `anioCopyright`. Cada constante `enum` en `Libro` en realidad es un objeto de tipo `Libro` que tiene su propia copia de las variables de instancia `titulo` y `anioCopyright`. El constructor (líneas 20 a 24) recibe dos parámetros `String`, uno que especifica el título del libro y otro que especifica el año de copyright del libro. Las líneas 22 y 23 asignan estos parámetros a las variables de instancia. Las líneas 27 a 36 declaran dos métodos, que devuelven el título del libro y el año de copyright, respectivamente.

La figura 8.11 prueba el tipo `enum` declarado en la figura 8.10 e ilustra cómo iterar a través de un rango de constantes `enum`. Para cada `enum`, el compilador genera el método `static` **values** (que se llama en la línea 12) que devuelve un arreglo de las constantes de `enum`, en el orden en el que se declararon. En la sección 7.6 vimos que la instrucción `for` mejorada puede usarse para iterar a través de un arreglo. Las líneas 12 a 14 utilizan la instrucción `for` mejorada para mostrar todas las constantes declaradas en la `enum` llamada `Libro`. La línea 14 invoca los métodos `obtenerTitulo` y `obtenerAnioCopyright` de `Libro` para obtener el título y el año de copyright asociados con la constante. Observe que cuando se convierte una constante `enum` en un objeto `String` (por ejemplo, `libro` en la línea 13), el identificador de la constante se utiliza como la representación `String` (por ejemplo, `JHTP6` para la primera constante `enum`).

```
1  // Fig. 8.11: PruebaEnum.java
2  // Prueba del tipo enum Libro.
3  import java.util.EnumSet;
4
5  public class PruebaEnum
6  {
7     public static void main( String args[] )
8     {
9        System.out.println( "Todos los libros:\n" );
10
11       // imprime todos los libros en enum Libro
12       for ( Libro libro : Libro.values() )
13          System.out.printf( "%-10s%-45s%s\n", libro,
14             libro.obtenerTitulo(), libro. obtenerAnioCopyright() );
15
16       System.out.println( "\nMostrar un rango de constantes enum:\n" );
17
18       // imprime los primeros cuatro libros
19       for ( Libro libro : EnumSet.range( Libro.JHTP6, Libro.CPPHTP4 ) )
20          System.out.printf( "%-10s%-45s%s\n", libro,
21             libro.obtenerTitulo(), libro.obtenerAnioCopyright() );
22    } // fin de main
23 } // fin de la clase PruebaEnum
```

```
Todos los libros:

JHTP6          Java How to Program 6e                                    2005
CHTP4          C How to Program 4e                                       2004
IW3HTP3        Internet & World Wide Web How to Program 3e               2004
CPPHTP4        C++ How to Program 4e                                     2003
VBHTP2         Visual Basic .NET How to Program 2e                       2002
CSHARPHTP      C# How to Program                                         2002

Mostrar un rango de constantes enum:

JHTP6          Java How to Program 6e                                    2005
CHTP4          C How to Program 4e                                       2004
IW3HTP3        Internet & World Wide Web How to Program 3e               2004
CPPHTP4        C++ How to Program 4e                                     2003
```

Figura 8.11 | Prueba de un tipo `enum`.

Las líneas 19 a 21 utilizan el método `static` **range** de la clase **EnumSet** (declarada en el paquete `java.util`) para mostrar un rango de las constantes de la `enum Libro`. El método `range` recibe dos parámetros (la primera y la última constantes `enum` en el rango) y devuelve un objeto `EnumSet` que contiene todas las constantes entre estas dos constantes. Por ejemplo, la expresión `EnumSet.range( Libro.JHTP6, Libro.CPPHTP4 )` devuelve un objeto `EnumSet` que contiene `Libro.JHTP6`, `Libro.CHTP4`, `Libro.IW3HTP3` y `Libro.CPPHTP4`. La instrucción `for` mejorada se puede utilizar con un objeto `EnumSet`, justo igual que como se utiliza con un arreglo, por lo que las líneas 19 a 21 utilizan la instrucción `for` mejorada para mostrar el título y el año de copyright de cada libro en el objeto `EnumSet`. La clase `EnumSet` proporciona varios métodos `static` más para crear conjuntos de constantes `enum` del mismo tipo de `enum`. Para obtener más detalles de la clase `EnumSet`, visite `java.sun.com/javase/6/docs/api/java/util/EnumSet.html`.



Error común de programación 8.6

En una declaración `enum`, es un error de sintaxis declarar constantes `enum` después de los constructores, campos y métodos del tipo de `enum` en su declaración.

8.10 Recolección de basura y el método finalize

Toda clase en Java tiene los métodos de la clase `Object` (paquete `java.lang`), uno de los cuales es el **método finalize**. Este método se utiliza raras veces. De hecho, buscamos a través de 6500 archivos de código fuente las clases de la API de Java, y encontramos menos de 50 declaraciones del método `finalize`. Sin embargo, y como `finalize` forma parte de cada clase, hablaremos aquí sobre este método para que a usted se le facilite comprender su propósito planeado, en caso de que se lo encuentre en sus estudios o en la industria. Los detalles completos acerca del método `finalize` están más allá del alcance de este libro, además de que la mayoría de los programadores no deben usarlo; pronto veremos por qué. Aprenderá más acerca de la clase `Object` en el capítulo 9, Programación orientada a objetos: herencia.

Todo objeto que creamos utiliza varios recursos del sistema, como la memoria. Para evitar "fugas de recursos", requerimos una manera disciplinada de regresar los recursos al sistema cuando ya no se necesitan. La Máquina Virtual de Java (JVM) realiza la **recolección automática de basura** para reclamar la memoria ocupada por los objetos que ya no se utilizan. Cuando ya no hay más referencias a un objeto, la JVM lo deja **marcado para la recolección de basura**. La memoria para dicho objeto se puede reclamar cuando la JVM ejecuta su **recolector de basura**, el cual es responsable de recuperar la memoria de los objetos que ya no se utilizan, para poder usarla con otros objetos. Por lo tanto, las fugas de memoria que son comunes en otros lenguajes como C y C++ (debido a que en esos lenguajes, la memoria no se reclama de manera automática) son menos probables en Java (pero algunas pueden ocurrir de todas formas, aunque con menos magnitud). Pueden ocurrir otros tipos de fugas de recursos. Por ejemplo, una aplicación podría abrir un archivo en disco para modificar el contenido. Si la aplicación no cierra el archivo, ninguna otra aplicación puede utilizarlo sino hasta que termine la aplicación que lo abrió.

El recolector de basura llama al método `finalize` para realizar las **tareas de preparación para la terminación** sobre un objeto, justo antes de que el recolector de basura reclame la memoria de ese objeto. El método `finalize` no recibe parámetros y tiene el tipo de valor de retorno `void`. Un problema con el método `finalize` es que no se garantiza que el recolector de basura se ejecute en un tiempo especificado. De hecho, tal vez el recolector de basura nunca se ejecute antes de que termine un programa. Por ende, no queda claro si (o cuándo) se hará la llamada al método `finalize`. Por esta razón, la mayoría de los programadores deben evitar el uso del método `finalize`. En la sección 8.11 demostraremos una situación en la que el recolector de basura llama al método `finalize`.



Observación de ingeniería de software 8.10

Una clase que utiliza recursos del sistema, como archivos en el disco, debe proporcionar un método para liberarlos en un momento dado. Muchas clases de la API de Java proporcionan métodos `close` o `dispose` para este propósito. Por ejemplo, la clase `Scanner` (`java.sun.com/javase/6/docs/api/java/util/Scanner.html`) tiene un método `close`.

8.11 Miembros de clase static

Cada objeto tiene su propia copia de todas las variables de instancia de la clase. En ciertos casos, sólo debe compartirse una copia de cierta variable entre todos los objetos de una clase. En esos casos se utiliza un **campo static**

(al cual se le conoce como una **variable de clase**). Una variable `static` representa **información en toda la clase** (todos los objetos de la clase comparten la misma pieza de datos). La declaración de una variable `static` comienza con la palabra clave `static`.

Veamos un ejemplo con datos `static`. Suponga que tenemos un videojuego con `Marcianos` y otras criaturas espaciales. Cada `Marciano` tiende a ser valiente y deseoso de atacar a otras criaturas espaciales cuando sabe que hay al menos otros cuatro `Marcianos` presentes. Si están presentes menos de cinco `Marcianos`, cada `Marciano` se vuelve cobarde. Por lo tanto, cada uno de ellos necesita saber el valor de `cuentaMarcianos`. Podríamos dotar a la clase `Marciano` con la variable `cuentaMarcianos` como variable de instancia. Si hacemos esto, entonces cada `Marciano` tendrá una copia separada de la variable de instancia, y cada vez que creemos un nuevo Marciano, tendremos que actualizar la variable de instancia `cuentaMarcianos` en todos los objetos `Marciano`. Las copias redundantes desperdician espacio y tiempo en actualizar cada una de las copias de la variable, además de ser un proceso propenso a errores. En vez de ello, declaramos a `cuentaMarcianos` como `static`, lo cual convierte a `cuentaMarcianos` en datos disponibles en toda la clase. Cada objeto `Marciano` puede ver la `cuentaMarcianos` como si fuera una variable de instancia de la clase `Marciano`, pero sólo se mantiene una copia de la variable `static cuentaMarcianos`. Esto nos ahorra espacio. Ahorramos tiempo al hacer que el constructor de `Marciano` incremente a la variable `static cuentaMarcianos`; como sólo hay una copia, no tenemos que incrementar cada una de las copias de `cuentaMarcianos` para cada uno de los objetos `Marciano`.



Observación de ingeniería de software 8.11

Use una variable `static` cuando todos los objetos de una clase tengan que utilizar la misma copia de la variable.

Las variables `static` tienen alcance a nivel de clase. Los miembros `public static` de una clase pueden utilizarse a través de una referencia a cualquier objeto de esa clase, o calificando el nombre del miembro con el nombre de la clase y un punto (`.`), como en `Math.random()`. Los miembros `private static` de una clase pueden utilizarse solamente a través de los métodos de esa clase. En realidad, los miembros `static` de una clase existen a pesar de que no existan objetos de esa clase; están disponibles tan pronto como la clase se carga en memoria, en tiempo de ejecución. Para acceder a un miembro `public static` cuando no existen objetos de la clase (y aún cuando sí existen), se debe anteponer el nombre de la clase y un punto (`.`) al miembro `static` de la clase, como en `Math.PI`. Para acceder a un miembro `private static` cuando no existen objetos de la clase debe proporcionarse un método `public static`, y para llamar a este método se debe calificar su nombre con el nombre de la clase y un punto.



Observación de ingeniería de software 8.12

Las variables de clase y los métodos `static` existen, y pueden utilizarse, incluso aunque no se hayan instanciado objetos de esa clase.

En nuestro siguiente programa declaramos dos clases: `Empleado` (figura 8.12) y `PruebaEmpleado` (figura 8.13). La clase `Empleado` declara a la variable `private static` llamada `cuenta` (figura 8.12, línea 9), y al método `public static` llamado `obtenerCuenta` (líneas 46 a 49). La variable `static cuenta` se inicializa con cero en la línea 9. Si no se inicializa una variable `static`, el compilador asigna a esa variable un valor predeterminado (en este caso, `0`). La variable `cuenta` mantiene la cuenta del número de objetos de la clase `Empleado` que residen actualmente en memoria. Esto incluye a los objetos que ya hayan sido marcados para la recolección de basura por la JVM, pero que el recolector de basura no ha reclamado todavía.

```
1  // Fig. 8.12: Empleado.java
2  // Variable static que se utiliza para mantener una cuenta del
3  // número de objetos Empleado en la memoria.
4
```

Figura 8.12 | Variable `static` que se utiliza para mantener cuenta del número de objetos `Empleado` en la memoria. (Parte 1 de 2).

```
5  public class Empleado
6  {
7     private String primerNombre;
8     private String apellidoPaterno;
9     private static int cuenta = 0; // número de objetos en memoria
10
11    // inicializa empleado, suma 1 a la variable static cuenta e
12    // imprime objeto String que indica que se llamó al constructor
13    public Empleado( String nombre, String apellido )
14    {
15       primerNombre = nombre;
16       apellidoPaterno = apellido;
17
18       cuenta++;  // incrementa la variable static cuenta de empleados
19       System.out.printf( "Constructor de Empleado: %s %s; cuenta = %d\n",
20          primerNombre, apellidoPaterno, cuenta );
21    } // fin de constructor de Empleado
22
23    // resta 1 a la variable static cuenta cuando el recolector
24    // de basura llama a finalize para borrar el objeto;
25    // confirma que se llamó a finalize
26    protected void finalize()
27    {
28       cuenta--; // decrementa la variable static cuenta de empleados
29       System.out.printf( "Finalizador de Empleado: %s %s; cuenta = %d\n",
30          primerNombre, apellidoPaterno, cuenta );
31    } // fin del método finalize
32
33    // obtiene el primer nombre
34    public String obtenerPrimerNombre()
35    {
36       return primerNombre;
37    } // fin del método obtenerPrimerNombre
38
39    // obtiene el apellido paterno
40    public String obtenerApellidoPaterno()
41    {
42       return apellidoPaterno;
43    } // fin del método obtenerApellidoPaterno
44
45    // método static para obtener el valor de la variable static cuenta
46    public static int obtenerCuenta()
47    {
48       return cuenta;
49    } // fin del método obtenerCuenta
50 } // fin de la clase Empleado
```

Figura 8.12 | Variable `static` que se utiliza para mantener cuenta del número de objetos `Empleado` en la memoria. (Parte 2 de 2).

Cuando existen objetos `Empleado`, el miembro `cuenta` se puede utilizar en cualquier método de un objeto `Empleado`; este ejemplo incrementa `cuenta` en el constructor (línea 18) y la decrementa en el método `finalize` (línea 28). Cuando no existen objetos de la clase `Empleado`, se puede hacer referencia de todas formas al miembro `cuenta`, pero sólo a través de una llamada al método `public static obtenerCuenta` (líneas 46 a 49), como en `Empleado.obtenerCuenta()`, lo cual devuelve el número de objetos `Empleado` que se encuentran actualmente en memoria. Cuando existen objetos, también se puede llamar el método `obtenerCuenta` a través de cualquier referencia a un objeto `Empleado`, como en la llamada `e1.obtenerCuenta()`.



Buena práctica de programación 8.1

Para invocar a cualquier método `static`, utilice el nombre de la clase y un punto (`.`) para enfatizar que el método que se está llamando es un método `static`.

Observe que la clase `Empleado` tiene un método `finalize` (líneas 26 a 31). Este método se incluye sólo para mostrar cuándo se ejecuta el recolector de basura en este programa. Por lo general, el método `finalize` se declara como `protected`, por lo que no forma parte de los servicios `public` de una clase. En el capítulo 9 hablaremos con detalle sobre el modificador de acceso de miembro `protected`.

El método `main` de `PruebaEmpleado` (figura 8.13) crea instancias de dos objetos `Empleado` (líneas 13 y 14). Cuando se invoca el constructor de cada objeto `Empleado`, en las líneas 15 y 16 de la figura 8.12 se asigna el primer nombre y el apellido paterno del `Empleado` a las variables de instancia `primerNombre` y `apellidoPaterno`. Observe que estas dos instrucciones no sacan copias de los argumentos `String` originales. En realidad, los objetos `String` en Java son inmutables (no pueden modificarse una vez que son creados). Por lo tanto, es seguro tener muchas referencias a un solo objeto `String`. Este no es normalmente el caso para los objetos de la mayoría de las otras clases en Java. Si los objetos `String` son inmutables, tal vez se pregunte por qué podemos utilizar los operadores `+` y `+=` para concatenar objetos `String`. En realidad, las operaciones de concatenación de objetos `String` producen un nuevo objeto `String`, el cual contiene los valores concatenados. Los objetos `String` originales no se modifican.

Cuando `main` ha terminado de usar los dos objetos `Empleado`, las referencias `e1` y `e2` se establecen en `null`, en las líneas 31 y 32. En este punto, las referencias `e1` y `e2` ya no hacen referencia a los objetos que se instanciaron en las líneas 13 y 14. Esto "marca a los objetos para la recolección de basura", ya que no existen más referencias a esos objetos en el programa.

```
1  // Fig. 8.13: PruebaEmpleado.java
2  // Demostración de miembros static.
3
4  public class PruebaEmpleado
5  {
6     public static void main( String args[] )
7     {
8        // muestra que la cuenta es 0 antes de crear Empleados
9        System.out.printf( "Empleados antes de instanciar: %d\n",
10          Empleado.obtenerCuenta() );
11
12       // crea dos Empleados; la cuenta debe ser 2
13       Empleado e1 = new Empleado( "Susan", "Baker" );
14       Empleado e2 = new Empleado( "Bob", "Blue" );
15
16       // muestra que la cuenta es 2 después de crear dos Empleados
17       System.out.println( "\nEmpleados despues de instanciar: " );
18       System.out.printf( "mediante e1.obtenerCuenta(): %d\n", e1.obtenerCuenta() );
19       System.out.printf( "mediante e2.obtenerCuenta(): %d\n", e2.obtenerCuenta() );
20       System.out.printf( "mediante Empleado.obtenerCuenta(): %d\n",
21          Empleado.obtenerCuenta() );
22
23       // obtiene los nombres de los Empleados
24       System.out.printf( "\nEmpleado 1: %s %s\nEmpleado 2: %s %s\n\n",
25          e1.obtenerPrimerNombre(), e1.obtenerApellidoPaterno(),
26          e2.obtenerPrimerNombre(), e2.obtenerApellidoPaterno() );
27
28       // en este ejemplo, sólo hay una referencia a cada Empleado,
29       // por lo que las siguientes dos instrucciones hacen que la JVM
30       // marque a cada objeto Empleado para la recolección de basura
31       e1 = null;
```

Figura 8.13 | Demostración de miembros `static`. (Parte 1 de 2).

```
32        e2 = null;
33
34        System.gc(); // pide que la recolección de basura se realice ahora
35
36        // muestra la cuenta de Empleados después de llamar al recolector de basura;
37        // la cuenta a mostrar puede ser 0, 1 o 2 dependiendo de si el recolector de
38        // basura se ejecuta de inmediato, y del número de objetos Empleado recolectados
39        System.out.printf( "\nEmpleados despues de System.gc(): %d\n",
40           Empleado.obtenerCuenta() );
41     } // fin de main
42  } // fin de la clase PruebaEmpleado
```

```
Empleados antes de instanciar: 0
Constructor de Empleado: Susan Baker; cuenta = 1
Constructor de Empleado: Bob Blue; cuenta = 2

Empleados despues de instanciar:
mediante e1.obtenerCuenta(): 2
mediante e2.obtenerCuenta(): 2
mediante Empleado.obtenerCuenta(): 2

Empleado 1: Susan Baker
Empleado 2: Bob Blue
Empleados despues de System.gc(): 2
Finalizador de Empleado: Susan Baker; cuenta = 0

Finalizador de Empleado: Bob Blue; cuenta = 1
```

Figura 8.13 | Demostración de miembros `static`. (Parte 2 de 2).

De un momento a otro, el recolector de basura podría reclamar la memoria para estos objetos (o el sistema operativo reclama la memoria cuando el programa termina). La JVM no garantiza cuándo se va a ejecutar el recolector de basura (o si acaso se va a ejecutar), por lo que este programa hace una llamada explícita al recolector de basura en la línea 34 (figura 8.13) utilizando el método `static` llamado `gc`, de la clase `System` (paquete `java.lang`) para indicar que el recolector de basura debe realizar su mejor esfuerzo por tratar de reclamar objetos que sean elegibles para la recolección de basura. Esto es sólo el mejor esfuerzo; es posible que no se recolecten objetos, o que se recolecte sólo un subconjunto de los objetos que sean candidatos. En los resultados de ejemplo de la figura 8.13, el recolector de basura se ejecutó antes de que en las líneas 39 y 40 se mostrara la cuenta actual de objetos `Empleado`. La última línea de la salida indica que el número de objetos `Empleado` en memoria es 0 después de la llamada a `System.gc()`. Además, las últimas dos líneas de la salida muestran que el objeto `Empleado` para `Bob Blue` se finalizó antes que el objeto `Empleado` para `Susan Baker`. Los resultados que obtenga en su sistema pueden ser distintos, ya que no se garantiza que el recolector de basura se ejecute al invocar a `System.gc()`, ni se garantiza que se recolecten los objetos en un orden específico.

[*Nota:* un método declarado como `static` no puede tener acceso a los miembros no `static` de una clase, ya que un método `static` puede llamarse aun cuando no se hayan creado instancias de objetos de la clase. Por la misma razón, esta referencia `this` no puede usarse en un método `static`; debe referirse a un objeto específico de la clase, y a la hora de llamar a un método `static`, podría no haber objetos de su clase en la memoria. La referencia `this` se requiere para permitir a un método de una clase acceder a otros miembros no `static` de la misma clase].



Error común de programación 8.7

Si un método `static` llama a un método de instancia (no `static`) en la misma clase utilizando sólo el nombre del método, se produce un error de compilación. De manera similar, se produce un error de compilación si un método `static` trata de acceder a una variable de instancia en la misma clase, utilizando sólo el nombre de la variable.



Error común de programación 8.8

Hacer referencia a `this` *en un método* `static` *es un error de sintaxis.*

8.12 Declaración `static import`

En la sección 6.3 aprendió acerca de los campos y métodos `static` de la clase `Math`. Para invocar a estos campos y métodos, anteponemos a cada uno de ellos el nombre de la clase `Math` y un punto (`.`). Una declaración **`static import`** nos permite hacer referencia a los miembros `static` importados, como si se hubieran declarado en la clase que los utiliza; el nombre de la clase y el punto (`.`) no se requieren para usar un miembro `static` importado.

Una declaración `static` import tiene dos formas: una que importa un miembro `static` específico (que se conoce como **declaración `static import` individual**) y una que importa a todos los miembros `static` de una clase (que se conoce como **declaración `static import` sobre demanda**). La siguiente sintaxis importa un miembro `static` específico:

`import static` *nombrePaquete.NombreClase.nombreMiembroEstático*`;`

en donde *nombrePaquete* es el paquete de la clase (por ejemplo, `java.lang`), *NombreClase* es el nombre de la clase (por ejemplo, `Math`) y *nombreMiembroEstático* es el nombre del campo o método `static` (por ejemplo, `PI` o `abs`). La siguiente sintaxis importa todos los miembros `static` de una clase:

`import static` *nombrePaquete.NombreClase*`.*;`

en donde *nombrePaquete* es el paquete de la clase (por ejemplo, `java.lang`) y *NombreClase* es el nombre de clase (por ejemplo, `Math`). El asterisco (`*`) indica que *todos* los miembros `static` de la clase especificada deben estar disponibles para usarlos en la(s) clase(s) declarada(s) en el archivo. Observe que las declaraciones `static import` sólo importan miembros de clase `static`. Las instrucciones `import` regulares deben usarse para especificar las clases utilizadas en un programa.

La figura 8.14 demuestra una declaración `static import`. La línea 3 es una declaración `static import`, la cual importa todos los campos y métodos `static` de la clase `Math`, del paquete `java.lang`. Las líneas 9 a 12 acceden al campo `static` llamado `E` (línea 11) de la clase `Math`, y los métodos `static sqrt` (línea 9), `ceil` (línea 10), `log` (línea 11) y `cos` (línea 12) sin anteponer el nombre de la clase `Math` y un punto al nombre del campo o a los nombres de los métodos.

```
1  // Fig. 8.14: PruebaStaticImport.java
2  // Uso de static import para importar métodos static de la clase Math.
3  import static java.lang.Math.*;
4
5  public class PruebaStaticImport
6  {
7     public static void main( String args[] )
8     {
9        System.out.printf( "sqrt( 900.0 ) = %.1f\n", sqrt( 900.0 ) );
10       System.out.printf( "ceil( -9.8 ) = %.1f\n", ceil( -9.8 ) );
11       System.out.printf( "log( E ) = %.1f\n", log( E ) );
12       System.out.printf( "cos( 0.0 ) = %.1f\n", cos( 0.0 ) );
13    } // fin de main
14 } // fin de la clase PruebaStaticImport
```

```
sqrt( 900.0 ) = 30.0
ceil( -9.8 ) = -9.0
log( E ) = 1.0
cos( 0.0 ) = 1.0
```

Figura 8.14 | Importación `static` de métodos de `Math`.



Error común de programación 8.9

Si un programa trata de importar métodos `static` *que tengan la misma firma, o campos* `static` *que tengan el mismo nombre, de dos o más clases, se produce un error de compilación.*

8.13 Variables de instancia final

El **principio del menor privilegio** es fundamental para la buena ingeniería de software. En el contexto de una aplicación, el principio establece que al código sólo se le debe otorgar tanto privilegio y acceso como necesite para llevar a cabo su tarea designada, pero no más. Veamos ahora cómo se aplica este principio a las variables de instancia.

Algunas variables de instancia necesitan modificarse, mientras que otras no. Usted puede utilizar la palabra clave `final` para especificar que una variable no puede modificarse (es decir, que sea una constante) y que cualquier intento por modificarla sería un error. Por ejemplo,

```
private final int INCREMENTO;
```

declara una variable de instancia `final` (constante) llamada `INCREMENTO`, de tipo `int`. Aunque las constantes se pueden inicializar al momento de declararse, no es obligatorio. Las constantes pueden inicializarse mediante cada uno de los constructores de la clase.



Observación de ingeniería de software 8.13

Declarar una variable de instancia como `final` *ayuda a hacer valer el principio del menor privilegio. Si una variable de instancia no debe modificarse, declárela como* `final` *para evitar su modificación.*

Nuestro siguiente ejemplo contiene dos clases: `Incremento` (figura 8.15) y `PruebaIncremento` (figura 8.16). La clase `Incremento` contiene una variable de instancia `final` de tipo `int`, llamada `INCREMENTO` (figura 8.15, línea 7). Observe que la variable `final` no se inicializa en su declaración, por lo que debe inicializarse mediante el constructor de la clase (líneas 9 a 13). Si la clase proporcionara varios constructores, cada constructor tendría que inicializar la variable `final`. El constructor recibe el parámetro `valorIncremento` de tipo `int` y asigna su valor a `INCREMENTO` (línea 12). Una variable `final` no puede modificarse mediante una asignación, una vez que se inicializa. La clase de aplicación `PruebaIncremento` crea un objeto de la clase `Incremento` (figura 8.16, línea 8), y proporciona el valor `5` como argumento para el constructor, el cual se asigna a la constante `INCREMENTO`.

```
1  // Fig. 8.15: Incremento.java
2  // variable de instancia final en una clase.
3
4  public class Incremento
5  {
6     private int total = 0; // el total de todos los incrementos
7     private final int INCREMENTO; // variable constante (sin inicializar)
8
9     // el constructor inicializa la variable de instancia final INCREMENTO
10    public Incremento( int valorIncremento )
11    {
12       INCREMENTO = valorIncremento; // inicializa la variable constante (una vez)
13    } // fin del constructor de Incremento
14
15    // suma INCREMENTO al total
16    public void sumarIncrementoATotal()
17    {
18       total += INCREMENTO;
19    } // fin del método sumarIncrementoATotal
20
21    // devuelve representación String de los datos de un objeto Incremento
```

Figura 8.15 | Variable de instancia `final` en una clase. (Parte 1 de 2).

```
    public String toString()
    {
       return String.format( "total = %d", total );
    } // fin del método toString
} // fin de la clase Incremento
```

Figura 8.15 | Variable de instancia `final` en una clase. (Parte 2 de 2).

```
// Fig. 8.16: PruebaIncremento.java
// variable final inicializada con el argumento de un constructor.

public class PruebaIncremento
{
   public static void main( String args[] )
   {
      Incremento valor = new Incremento( 5 );

      System.out.printf( "Antes de incrementar: %s\n\n", valor );

      for ( int i = 1; i <= 3; i++ )
      {
         valor.sumarIncrementoATotal();
         System.out.printf( "Después de incrementar %d: %s\n", i, valor );
      } // fin de for
   } // fin de main
} // fin de la clase PruebaIncremento
```

```
Antes de incrementar: total = 0

Despues de incrementar 1: total = 5
Despues de incrementar 2: total = 10
Despues de incrementar 3: total = 15
```

Figura 8.16 | Variable `final` inicializada con el argumento de un constructor.



Error común de programación 8.10

Tratar de modificar una variable de instancia `final` *después de inicializarla es un error de compilación.*



Tip para prevenir errores 8.2

Los intentos por modificar una variable de instancia `final` *se atrapan en tiempo de compilación, en vez de producir errores en tiempo de ejecución. Esto siempre es preferible en vez de permitir que se pasen hasta el tiempo de ejecución (en donde los estudios han demostrado que la reparación es, a menudo, mucho más costosa).*



Observación de ingeniería de software 8.14

Un campo `final` *también debe declararse como* `static`*, si se inicializa en su declaración. Una vez que se inicializa un campo* `static` *en su declaración, su valor ya no puede cambiar. Por lo tanto, no es necesario tener una copia separada del campo para cada objeto de la clase. Al hacer a ese campo* `static`*, se permite que todos los objetos de la clase compartan el campo* `final`*.*

Si no se inicializa una variable `final`, se produce un error de compilación. Para demostrar esto, colocamos la línea 12 de la figura 8.15 en un comentario y recompilamos la clase. La figura 8.17 muestra el mensaje de error que produce el compilador.



Error común de programación 8.11

Si no se inicializa una variable de instancia `final` *en su declaración, o en cada constructor de la clase, se produce un error de compilación, indicando que la variable tal vez no se haya inicializado. El mismo error se produce si la clase inicializa la variable en algunos de los constructores de la clase, pero no en todos.*

```
Incremento.java:13: variable INCREMENTO might not have been initialized
    } // fin del constructor de Incremento
    ^
1 error
```

Figura 8.17 | La variable `final INCREMENTO` debe inicializarse.

8.14 Reutilización de software

Los programadores en Java se concentran en fabricar nuevas clases y reutilizar las ya existentes. Existen muchas bibliotecas de clases, y se están desarrollando más a nivel mundial. El software se construye entonces a partir de componentes existentes, bien definidos, cuidadosamente probados, bien documentados, portables y ampliamente utilizados. Este tipo de reutilización de software agiliza el desarrollo de software poderoso de alta calidad. El **desarrollo rápido de aplicaciones (RAD)** es de gran interés hoy en día.

Hay miles de clases a escoger en la API de Java, para ayudarnos a implementar programas. Evidentemente, Java no es tan solo un lenguaje de programación. Es un marco de trabajo en el que los desarrolladores lograr una verdadera reutilización y un desarrollo rápido de aplicaciones. Los programadores pueden enfocarse en la tarea principal al desarrollar sus programas, y dejan los detalles de nivel inferior a las clases de la API de Java. Por ejemplo, para escribir un programa que dibuja gráficos, un programador no requiere de un conocimiento sobre los gráficos en todas las plataformas computacionales en las que el programa vaya a ejecutarse. En vez de ello, puede concentrarse en aprender las herramientas para gráficos en Java (que son bastante substanciales, y cada día aumentan más) y escribir un programa en Java que dibuje los gráficos, utilizando clases de la API, como `Graphics`. Cuando el programa se ejecuta en cierta computadora, corresponde a la JVM traducir los comandos de Java en comandos que la computadora local pueda entender.

Las clases de la API de Java permiten a los programadores llevar con más rapidez nuevas aplicaciones al mercado, mediante el uso de componentes preexistentes y comprobados. Esto no sólo reduce el tiempo de desarrollo, sino que también mejora la habilidad del programador en cuanto a depurar y mantener aplicaciones. Para aprovechar las diversas herramientas de Java, es imprescindible que usted se familiarice con la variedad de paquetes y clases en la API de Java. En el sitio `java.sun.com` existen muchos recursos basados en Web, los cuales le pueden ayudar con esta tarea. El principal recurso para aprender acerca de la API de Java es la documentación de la misma, que puede encontrarse en

```
java.sun.com/javase/6/docs/api/
```

Puede descargar la documentación de la API en

```
java.sun.com/javase/downloads/ea.jsp
```

Además, `java.sun.com` proporciona muchos otros recursos, incluyendo tutoriales, artículos y sitios específicos acerca de temas específicos de Java.



Buena práctica de programación 8.2

Evite reinventar la rueda. Estudie las capacidades de la API de Java. Si la API contiene una clase que cumple con los requerimientos de su programa, utilícela en lugar de una.

Para darnos cuenta de todo el potencial de la reutilización de software, necesitamos mejorar los esquemas de catalogación, esquemas de licenciamiento, los mecanismos de protección que aseguran que no se corrompan las copias maestras de las clases, esquemas de descripción que los diseñadores de sistemas utilizan para determinar si

los objetos existentes cumplen con sus necesidades, mecanismos de exploración que determinan cuáles clases están disponibles y qué tan estrechamente cumplen con los requerimientos de los desarrolladores de software, etcétera. Muchos problemas interesantes de investigación y desarrollo se han resuelto, y muchos más necesitan resolverse. Estos problemas se resolverán, debido a que el valor potencial de la reutilización de software es enorme.

8.15 Abstracción de datos y encapsulamiento

Las clases normalmente ocultan los detalles de la implementación a los clientes. Esto se conoce como **ocultamiento de información**. Como ejemplo de ello, analicemos la estructura de datos llamada **pila**, que presentamos en la sección 6.6. Recuerde que una pila es una estructura de datos del tipo **último en entrar, primero en salir (UEPS)**: el último elemento que se mete (inserta) en la pila es el primer elemento que se saca (extrae) de ella.

Las pilas pueden implementarse mediante arreglos y con otras estructuras de datos, como las listas enlazadas. (Hablaremos sobre las pilas y las listas enlazadas en el capítulo 17, Estructuras de datos, y en el capítulo 19, Colecciones). El cliente de una clase pila no necesita preocuparse por la implementación de la pila. El cliente sólo sabe que cuando se colocan elementos de datos en la pila, éstos se recuperarán en el orden del último en entrar, primero en salir. El cliente se preocupa acerca de qué funcionalidad ofrece una pila, pero no acerca de cómo se implementa esa funcionalidad. A este concepto se le conoce como **abstracción de datos**. Aunque los programadores pudieran conocer los detalles de la implementación de una clase, no deben escribir código que dependa de esos detalles. Esto permite que una clase en particular (como una que implemente a una pila y sus operaciones: *meter* y *sacar*) se reemplace con otra versión, sin afectar al resto del sistema. Mientras que los servicios `public` de la clase no cambien (es decir, que cada método original tenga aún el mismo nombre, tipo de valor de retorno y lista de parámetros en la declaración de la nueva clase), el resto del sistema no se ve afectado.

La mayoría de los lenguajes de programación enfatizan las acciones. En estos lenguajes, los datos existen para apoyar las acciones que los programas deben realizar. Los datos son "menos interesantes" que las acciones. Los datos son "crudos". Sólo existen unos cuantos tipos primitivos, y es difícil para los programadores crear sus propios tipos. Java y el estilo orientado a objetos de programación elevan la importancia de los datos. Las principales actividades de la programación orientada a objetos en Java son la creación de tipos (por ejemplo, clases) y la expresión de las interacciones entre objetos de esos tipos. Para crear lenguajes que enfaticen los datos, la comunidad de lenguajes de programación necesitaba formalizar ciertas nociones sobre los datos. La formalización que consideramos aquí es la noción de **tipos de datos abstractos (ADTs)**, los cuales mejoran el proceso de desarrollo de software.

Considere el tipo primitivo `int`, el cual la mayor parte de las personas lo asociarían con un entero en matemáticas. En vez de ello, un `int` es una representación abstracta de un entero. A diferencia de los enteros matemáticos, los números `int` de computadora tienen un tamaño fijo. Por ejemplo, el tipo `int` en Java está limitado al rango desde –2,147,483,648 hasta +2,147,483,647. Si el resultado de un cálculo queda fuera de este rango se produce un error, y la computadora responde en cierta forma dependiente del equipo. Por ejemplo, podría producir "silenciosamente" un resultado incorrecto, como un valor demasiado largo como para caber en una variable `int` (lo que comúnmente se conoce como **desbordamiento aritmético**). Los enteros matemáticos no tienen este problema. Por lo tanto, la noción de un `int` de computadora es solamente una aproximación de la noción de un entero real. Lo mismo se aplica al tipo `float` y a los demás tipos integrados.

Hemos dado por sentado la noción de `int` hasta este punto, pero ahora analicémosla desde una nueva perspectiva. Los tipos como `int`, `float` y `char` son ejemplos de tipos de datos abstractos. Estos tipos son representaciones de nociones reales hasta cierto nivel satisfactorio de precisión, dentro de un sistema computacional.

En realidad, un ADT captura dos nociones: una **representación de datos** y las **operaciones** que pueden realizarse sobre esos datos. Por ejemplo, en Java un `int` contiene un valor entero (datos) y proporciona las operaciones de suma, resta, multiplicación, división y residuo; sin embargo, la división entre cero está indefinida. Los programadores en Java utilizan clases para implementar tipos de datos abstractos.



Observación de ingeniería de software 8.15

Los programadores pueden crear tipos mediante el uso del mecanismo de clases. Pueden diseñarse nuevos tipos de manera que sean tan convenientes de usar como los tipos integrados. Esto marca a Java como un lenguaje extensible. Aunque el lenguaje es fácil de extender mediante nuevos tipos, el programador no puede alterar el lenguaje básico en sí.

Otro de los tipos de datos abstractos que veremos es una **cola**, similar a una "línea de espera". Los sistemas computacionales utilizan muchas colas internamente. Una cola ofrece un comportamiento bien definido a sus clientes: éstos colocan elementos en una cola, uno a la vez, mediante una operación conocida como *enfilar*, y luego recuperan esos elementos, uno a la vez, mediante una operación conocida como *retirar*. Una cola devuelve los elementos en el orden **primero en entrar, primero en salir (PEPS)**, lo cual significa que el primer elemento insertado en una cola es el primer elemento que se remueve. Conceptualmente, una cola puede volverse infinitamente larga, pero las colas reales son finitas.

La cola oculta una representación interna de datos que lleva el registro de los elementos que esperan actualmente en la línea, y ofrece operaciones a sus clientes (*enfilar* y *retirar*). A los clientes no les preocupa la implementación de la cola; simplemente dependen de que ésta opere "como se indicó". Cuando un cliente enfila a un elemento, la cola debe aceptarlo y colocarlo en algún tipo de estructura de datos PEPS interna. De manera similar, cuando el cliente desea el siguiente elemento de la parte frontal de la cola, ésta debe remover el elemento de su representación interna y entregarlo en orden PEPS (es decir, el elemento que haya estado más tiempo en la cola debe ser el siguiente que se devuelva mediante la siguiente operación de retiro).

El ADT tipo cola garantiza la integridad de su estructura de datos interna. Los clientes no pueden manipular esta estructura de datos directamente; sólo el ADT tipo cola tiene acceso a sus datos internos. Los clientes pueden realizar solamente las operaciones permitidas en la representación de datos; el ADT rechaza las operaciones que su interfaz pública no proporciona.

8.16 Ejemplo práctico de la clase `Tiempo`: creación de paquetes

En casi todos los ejemplos de este libro hemos visto que las clases de bibliotecas preexistentes, como la API de Java, pueden importarse en un programa en Java. Cada clase en la API pertenece a un paquete que contiene un grupo de clases relacionadas. A medida que las aplicaciones se vuelven más complejas, los paquetes ayudan a los programadores a administrar la complejidad de los componentes de una aplicación. Los paquetes también facilitan la reutilización de software, al permitir que los programas importen clases de otros paquetes (como lo hemos hecho en la mayoría de los ejemplos). Otro beneficio de los paquetes es que proporcionan una convención para los nombres de clases únicos, lo cual ayuda a evitar los conflictos de nombres de clases (que veremos más adelante). En esta sección veremos cómo crear sus propios paquetes.

Pasos para declarar una clase reutilizable

Antes de poder importar una clase en varias aplicaciones, ésta debe colocarse en un paquete para que sea reutilizable. La figura 8.18 muestra cómo especificar el paquete en el que debe colocarse una clase. La figura 8.19 muestra cómo importar nuestra clase empaquetada, para poder usarla en una aplicación. Los pasos para crear una clase reutilizable son:

1. Declare una clase `public`; ya que de lo contrario, sólo la podrán usar otras clases en el mismo paquete.
2. Seleccione un nombre único para el paquete y agregue una **declaración `package`** al archivo de código fuente para la declaración de la clase reutilizable. Sólo puede haber una declaración `package` en cada archivo de código fuente de Java, y debe ir antes que todas las demás declaraciones e instrucciones en el archivo. Observe que los comentarios no son instrucciones, por lo que pueden colocarse antes de una instrucción `package` en un archivo.
3. Compile la clase de manera que se coloque en la estructura de directorio del paquete apropiada.
4. Importe la clase reutilizable en un programa, y utilícela.

Pasos 1 y 2: crear una clase `public` *y agregar la instrucción* `package`

Para el *paso 1*, modificaremos la clase `public Tiempo1` que declaramos en la figura 8.1. La nueva versión se muestra en la figura 8.18. No se han hecho modificaciones a la implementación de la clase, por lo que no hablaremos otra vez aquí sobre sus detalles de implementación.

Para el *paso 2* agregamos una declaración `package` (línea 3), la cual declara a un paquete llamado `com.deitel.jhtp7.cap08`. Al colocar una declaración `package` al principio de un archivo de código fuente de Java, indicamos que la clase declarada en el archivo forma parte del paquete especificado. Sólo las declaraciones

```
// Fig. 8.18: Tiempo1.java
// La declaración de la clase Tiempo1 mantiene la hora en formato de 24 horas.
package com.deitel.jhtp7.cap08;

public class Tiempo1
{
   private int hora;    // 0 - 23
   private int minuto;  // 0 - 59
   private int segundo; // 0 - 59

   // establece un nuevo valor de tiempo, usando la hora universal; asegura que
   // los datos sean consistentes, al establecer los valores inválidos a cero
   public void establecerTiempo( int h, int m, int s )
   {
      hora = ( ( h >= 0 && h < 24 ) ? h : 0 );    // valida la hora
      minuto = ( ( m >= 0 && m < 60 ) ? m : 0 );  // valida el minuto
      segundo = ( ( s >= 0 && s < 60 ) ? s : 0 ); // valida el segundo
   } // fin del método establecerTiempo

   // convierte a objeto String en formato de hora universal (HH:MM:SS)
   public String aStringUniversal()
   {
      return String.format( "%02d:%02d:%02d", hora, minuto, segundo );
   } // fin del método aStringUniversal

   // convierte a objeto String en formato de hora estándar (H:MM:SS AM or PM)
   public String toString()
   {
      return String.format( "%d:%02d:%02d %s",
         ( ( hora == 0 || hora == 12 ) ? 12 : hora % 12 ),
         minuto, segundo, ( hora < 12 ? "AM" : "PM" ) );
   } // fin del método toString
} // fin de la clase Tiempo1
```

Figura 8.18 | Empaquetamiento de la clase `Tiempo1` para reutilizarla.

`package`, las declaraciones `import` y los comentarios pueden aparecer fuera de las llaves de una declaración de clase. Un archivo de código fuente de Java debe tener el siguiente orden:

1. Una declaración `package` (si la hay).
2. Declaraciones `import` (si las hay).
3. Declaraciones de clases.

Sólo una de las declaraciones de las clases en un archivo específico pueden ser `public`; las demás se colocan en el paquete, y sólo las pueden utilizar las otras clases en el mismo paquete. Las clases que no son `public` están en un paquete, para dar soporte a las clases reutilizables.

En un esfuerzo por proporcionar nombres únicos para cada paquete, Sun Microsystems especifica una convención para nombrar paquetes, que todos los programadores de Java deben seguir. Cada nombre de paquete debe empezar con un nombre de dominio de Internet en orden inverso. Por ejemplo, nuestro nombre de dominio es `deitel.com`, por lo que los nombres de nuestros paquetes empiezan con `com.deitel`. Para el nombre de dominio *suescuela*`.edu`, el nombre del paquete debe empezar con `edu.`*suescuela*. Una vez que se invierte el nombre del dominio, podemos elegir cualquier otro nombre que deseemos para nuestro paquete. Si usted forma parte de una empresa con muchas divisiones, o de una universidad con muchas escuelas, tal vez sea conveniente que utilice el nombre de su división o escuela como el siguiente nombre en el paquete. Nosotros optamos por usar `jhtp7` como el siguiente nombre en nuestro paquete, para indicar que esta clase es del libro en inglés *Java How To Program, Séptima edición*. El último nombre en nuestro paquete especifica que es para el capítulo 8 (`cap08`).

Paso 3: compilar la clase empaquetada

El *paso 3* es compilar la clase, de manera que se almacene en el paquete apropiado. Cuando se compila un archivo de Java que contiene una declaración `package`, el archivo de clase resultante se coloca en el directorio especificado por la declaración. La declaración `package` en la figura 8.18 indica que la clase `Tiempo1` debe colocarse en el siguiente directorio:

```
com
   deitel
      jhtp7
         cap08
```

Los nombres de los directorios especifican la ubicación exacta de las clases en el paquete.

Al compilar una clase en un paquete, la opción **-d** de la línea de comandos de `javac` hace que el compilador `javac` cree los directorios apropiados, con base en la declaración `package` de la clase. Esta opción también especifica en dónde se deben almacenar los directorios. Por ejemplo, en una ventana de comandos utilizamos el siguiente comando de compilación

```
javac -d . Tiempo1.java
```

para especificar que el primer directorio en el nombre de nuestro paquete debe colocarse en el directorio actual. El punto (`.`) después de `-d` en el comando anterior representa el directorio actual en los sistemas operativos Windows, UNIX y Linux (y en varios otros también). Después de ejecutar el comando de compilación, el directorio actual contiene un directorio llamado `com`, el cual contiene uno llamado `deitel`, que a su vez contiene uno llamado `jhtp7`, y este último contiene un directorio llamado `cap08`. En el directorio `cap08` podemos encontrar el archivo `Tiempo1.class`. [*Nota:* si no utiliza la opción `-d`, entonces debe copiar o mover el archivo de clase al directorio del paquete apropiado después de compilarlo].

El nombre `package` forma parte del nombre de clase completamente calificado, por lo que el nombre de la clase `Tiempo1` es en realidad `com.deitel.jhtp7.cap08.Tiempo1`. Puede utilizar este nombre completamente calificado en sus programas, o puede importar la clase y utilizar su **nombre simple** (el nombre de la clase por sí solo: `Tiempo1`) en el programa. Si otro paquete contiene también una clase `Tiempo1`, los nombres de clase completamente calificados pueden utilizarse para diferenciar una clase de otra en el programa, y evitar un **conflicto de nombres** (también conocido como **colisión de nombres**).

Paso 4: importar la clase reutilizable

Una vez que la clase se compila y se guarda en su paquete, se puede importar en los programas (*paso 4*). En la aplicación `PruebaPaqueteTiempo1` de la figura 8.19, la línea 3 especifica que la clase `Tiempo1` debe importarse para usarla en la clase `PruebaPaqueteTiempo1`. La clase `PruebaPaqueteTiempo1` está en el paquete predeterminado, ya que el archivo `.java` de la clase no contiene una declaración `package`. Como las dos clases se encuentran en distintos paquetes, se requiere la declaración `import` en la línea 3, de manera que la clase `PruebaPaqueteTiempo1` pueda utilizar la clase `Tiempo1`.

```
1  // Fig. 8.19: PruebaPaqueteTiempo1.java
2  // Uso de un objeto Tiempo1 en una aplicación.
3  import com.deitel.jhtp7.cap08.Tiempo1; // importa la clase Tiempo1
4
5  public class PruebaPaqueteTiempo1
6  {
7     public static void main( String args[] )
8     {
9        // crea e inicializa un objeto Tiempo1
10       Tiempo1 tiempo = new Tiempo1(); // llama al constructor de Tiempo1
11
12       // imprime representaciones String de la hora
13       System.out.print( "La hora universal inicial es: " );
```

Figura 8.19 | Uso de un objeto `Tiempo1` en una aplicación. (Parte 1 de 2).

```
14         System.out.println( tiempo.aStringUniversal() );
15         System.out.print( "La hora estandar inicial es: " );
16         System.out.println( tiempo.toString() );
17         System.out.println(); // imprime una línea en blanco
18
19         // cambia la hora e imprime la hora actualizada
20         tiempo.establecerTiempo( 13, 27, 6 );
21         System.out.print( "La hora universal despues de establecerTiempo es: " );
22         System.out.println( tiempo.aStringUniversal() );
23         System.out.print( "La hora estandar despues de establecerTiempo es: " );
24         System.out.println( tiempo.toString() );
25         System.out.println(); // imprime una línea en blanco
26
27         // establece la hora con valores inválidos; imprime la hora actualizada
28         tiempo.establecerTiempo( 99, 99, 99 );
29         System.out.println( "Despues de intentar ajustes invalidos:" );
30         System.out.print( "Hora universal: " );
31         System.out.println( tiempo.aStringUniversal() );
32         System.out.print( "Hora estandar: " );
33         System.out.println( tiempo.toString() );
34      } // fin de main
35   } // fin de la clase PruebaPaqueteTiempo1
```

```
La hora universal inicial es: 00:00:00
La hora estandar inicial es: 12:00:00 AM

La hora universal despues de establecerTiempo es: 13:27:06
La hora estandar despues de establecerTiempo es: 1:27:06 PM

Despues de intentar ajustes invalidos:
Hora universal: 00:00:00
Hora estandar: 12:00:00 AM
```

Figura 8.19 | Uso de un objeto `Tiempo1` en una aplicación. (Parte 2 de 2).

La línea 3 se conoce como una **declaración `import` de tipo simple**; es decir, la declaración `import` especifica una clase que se va a importar. Cuando su programa utiliza varias clases del mismo paquete, puede importar esas clases con una sola declaración `import`. Por ejemplo, la declaración

```
import java.util.*; // importa las clases del paquete java.util
```

usa un asterisco (*) al final de la declaración `import` para informar al compilador que todas las clases del paquete `java.util` están disponibles para usarlas en el programa. Esto se conoce como una **declaración `import` tipo sobre demanda**. La JVM sólo carga las clases del paquete `java.util` que se utilizan en el programa. La declaración `import` anterior nos permite utilizar el nombre simple de cualquier clase del paquete `java.util` en el programa. A lo largo de este libro, utilizaremos declaraciones `import` tipo simples, por claridad.



Error común de programación 8.12

Utilizar la declaración `import java.*;` produce un error de compilación. Se debe especificar el nombre exacto del paquete del que se desea importar clases.

Especificar la ruta de clases durante la compilación

Al compilar `PruebaPaqueteTiempo1`, `javac` debe localizar el archivo `.class` para `Tiempo1`, de forma que se asegure que la clase `PruebaPaqueteTiempo1` utilice a la clase `Tiempo1` en forma correcta. El compilador utiliza un objeto especial, llamado **cargador de clases**, para localizar las clases que necesita. El cargador de clases em-

pieza buscando las clases estándar de Java que se incluyen con el JDK. Después busca los **paquetes opcionales**. Java cuenta con un **mecanismo de extensión** que permite agregar paquetes nuevos (opcionales), para fines de desarrollo y ejecución. [*Nota:* el mecanismo de extensión está más allá del alcance de este libro. Para obtener más información, visite `java.sun.com/javase/6/docs/technotes/guides/extensions/`]. Si la clase no se encuentra en las clases estándar de Java o en las clases de extensión, el cargador de clases busca en la **ruta de clases**, que contiene una lista de ubicaciones en la que se almacenan las clases. La ruta de clases consiste en una lista de directorios o **archivos de ficheros**, cada uno separado por un **separador de directorio**: un signo de punto y coma (`;`) en Windows o un signo de dos puntos (`:`) en UNIX/Linux/Mac OS X. Los archivos de ficheros son archivos individuales que contienen directorios de otros archivos, generalmente en formato comprimido. Por ejemplo, las clases estándar de Java que usted utiliza en sus programas están contenidas en el archivo de ficheros `rt.jar`, el cual se instala junto con el JDK. Los archivos de ficheros generalmente terminan con la extensión `.jar` o `.zip`. Los directorios y archivos de ficheros que se especifican en la ruta de clases contienen las clases que usted desea poner a disponibilidad del compilador y la máquina virtual de Java.

De manera predeterminada, la ruta de clases consiste sólo del directorio actual. Sin embargo, la ruta de clases puede modificarse de la siguiente manera:

1. proporcionando la opción **–classpath** al compilador `javac` o
2. estableciendo la variable de entorno `CLASSPATH` (una variable especial que usted define y el sistema operativo mantiene, de manera que las aplicaciones puedan buscar clases en las ubicaciones especificadas).

Para obtener más información sobre la ruta de clases, visite la página `java.sun.com/javase/6/docs/technotes/tools/index.html`. La sección titulada "General Information" (información general) contiene información acerca de cómo establecer la ruta de clases para UNIX/Linux y Windows.



Error común de programación 8.13

Al especificar una ruta de clases explícita se elimina el directorio actual de la ruta de clases. Esto evita que las clases en el directorio actual (incluyendo los paquetes en ese directorio) se carguen correctamente. Si deben cargarse clases del directorio actual, un punto (`.`) en la ruta de clases para especificar el directorio actual.



Observación de ingeniería de software 8.16

En general, es una mejor práctica utilizar la opción `–classpath` del compilador, en vez de usar la variable de entorno `CLASSPATH` para especificar la ruta de clases para un programa. De esta manera, cada aplicación puede tener su propia ruta de clases.



Tip para prevenir errores 8.3

Al especificar la ruta de clases con la variable de entorno `CLASSPATH` se pueden producir errores sutiles y difíciles de localizar en los programas que utilicen versiones distintas del mismo paquete.

Para el ejemplo de las figuras 8.18 y 8.19, no especificamos una ruta de clases explícita. Por lo tanto, para localizar las clases en el paquete `com.deitel.jhtp7.cap08` de este ejemplo, el cargador de clases busca en el directorio actual el primer nombre en el paquete: `com`. A continuación, el cargador de clases navega por la estructura de directorios. El directorio `com` contiene al subdirectorio `deitel`; éste contiene al subdirectorio `jhtp7`. Finalmente, el directorio `jhtp7` contiene al subdirectorio `cap08`. En este subdirectorio se encuentra el archivo `Tiempo1.class`, que se carga mediante el cargador de clases para asegurar que la clase se utilice apropiadamente en nuestro programa.

Especificar la ruta de clases al ejecutar una aplicación

Al ejecutar una aplicación, la JVM debe poder localizar las clases que se utilizan en esa aplicación. Al igual que el compilador, el comando `java` utiliza un cargador de clases que busca primero en las clases estándar y de extensión, y después busca en la ruta de clases (el directorio actual, de manera predeterminada). La ruta de clases para la JVM puede especificarse en forma explícita, utilizando cualquiera de las técnicas descritas para el compilador. Al igual que con el compilador, es mejor especificar a la JVM una ruta de clases individual para cada programa, mediante las opciones de la línea de comandos. Usted puede especificar la ruta de clases en el comando `java`

mediante las opciones de línea de comandos `-classpath` o `-cp`, seguidas de una lista de directorios o archivos de ficheros separados por signos de punto y coma (`;`) en Microsoft Windows, o signos de dos puntos (`:`) en UNIX/Linux/Mac OS X. De nuevo, si las clases deben cargarse del directorio actual, asegúrese de incluir un punto (`.`) en la ruta de clases para especificar el directorio actual.

8.17 Acceso a paquetes

Si no se especifica un modificador de acceso (`public`, `protected` o `private`; hablaremos sobre `protected` en el capítulo 9) para un método o variable al declararse en una clase, se considerará que el método o variable tiene **acceso a nivel de paquete**. En un programa que consiste de una declaración de clase, esto no tiene un efecto específico. No obstante, si un programa utiliza varias clases del mismo paquete (es decir, un grupo de clases relacionadas), éstas pueden acceder a los miembros con acceso a nivel de paquete de cada una de las otras clases directamente, a través de referencias a objetos de las clases apropiadas.

La aplicación de la figura 8.20 demuestra el acceso a los paquetes. La aplicación contiene dos clases en un archivo de código fuente: la clase de aplicación `PruebaDatosPaquete` (líneas 5 a 21) y la clase `DatosPaquete` (líneas 24 a 41). Al compilar este programa, el compilador produce dos archivos `.class` separados: `PruebaDatosPaquete.class` y `DatosPaquete.class`. El compilador coloca los dos archivos `.class` en el mismo directorio, por lo que las clases se consideran como parte del mismo paquete. Como forman parte del mismo paquete, se permite a la clase `PruebaDatosPaquete` modificar los datos con acceso a nivel de paquete de los objetos `DatosPaquete`.

En la declaración de la clase `DatosPaquete`, las líneas 26 y 27 declaran las variables de instancia `numero` y `cadena` sin modificadores de acceso; por lo tanto, éstas son variables de instancia con acceso a nivel de paquete. El método `main` de la aplicación `PruebaDatosPaquete` crea una instancia de la clase `DatosPaquete` (línea 9) para demostrar la habilidad de modificar las variables de instancia de `DatosPaquete` directamente (como se muestra en las líneas 15 y 16). Los resultados de la modificación se pueden ver en la ventana de resultados.

```
1  // Fig. 8.20: PruebaDatosPaquete.java
2  // Los miembros con acceso a nivel de paquete de una clase son accesibles
3  // para las demás clases en el mismo paquete.
4
5  public class PruebaDatosPaquete
6  {
7     public static void main( String args[] )
8     {
9        DatosPaquete datosPaquete = new DatosPaquete();
10
11       // imprime la representación String de datosPaquete
12       System.out.printf( "Despues de instanciar:\n%s\n", datosPaquete );
13
14       // modifica los datos con acceso a nivel de paquete en el objeto datosPaquete
15       datosPaquete.numero = 77;
16       datosPaquete.cadena = "Adios";
17
18       // imprime la representación String de datosPaquete
19       System.out.printf( "\nDespues de modificar valores:\n%s\n", datosPaquete );
20    } // fin de main
21 } // fin de la clase PruebaDatosPaquete
22
23 // clase con variables de instancia con acceso a nivel de paquete
24 class DatosPaquete
25 {
26    int numero; // variable de instancia con acceso a nivel de paquete
27    String cadena; // variable de instancia con acceso a nivel de paquete
```

Figura 8.20 | Los miembros con acceso a nivel de paquete de una clase son accesibles para las demás clases en el mismo paquete. (Parte 1 de 2).

```

   // constructor
   public DatosPaquete()
   {
      numero = 0;
      cadena = "Hola";
   } // fin del constructor de DatosPaquete

   // devuelve la representación String del objeto DatosPaquete
   public String toString()
   {
      return String.format( "numero: %d; cadena: %s", numero, cadena );
   } // fin del método toString
} // fin de la clase DatosPaquete
```

```
Despues de instanciar:
numero: 0; cadena: Hola

Despues de modificar valores:
numero: 77; cadena: Adios
```

Figura 8.20 | Los miembros con acceso a nivel de paquete de una clase son accesibles para las demás clases en el mismo paquete. (Parte 2 de 2).

8.18 (Opcional) Ejemplo práctico de GUI y gráficos: uso de objetos con gráficos

La mayoría de los gráficos que ha visto hasta este punto no varían cada vez que se ejecuta el programa. Sin embargo, el ejercicio 6.2 le pedía que creara un programa para generar figuras y colores al azar. En ese ejercicio, el dibujo cambiaba cada vez que el sistema llamaba a `paintComponent` para volver a dibujar el panel. Para crear un dibujo más consistente que permanezca sin cambios cada vez que se dibuja, debemos almacenar información acerca de las figuras mostradas, para que podamos reproducirlas en forma idéntica, cada vez que el sistema llame a `paintComponent`.

Para ello, crearemos un conjunto de clases de figuras que almacenan información acerca de cada figura. Haremos a estas clases "inteligentes", al permitir que los objetos se dibujen a sí mismos si se les proporciona un objeto `Graphics`. La figura 8.21 declara la clase `MiLinea`, que tiene todas estas capacidades.

La clase `MiLinea` importa a `Color` y `Graphics` (líneas 3 y 4). Las líneas 8 a 11 declaran variables de instancia para las coordenadas necesarias para dibujar una línea, y la línea 12 declara la variable de instancia que almacena el color. El constructor en las líneas 15 a 22 recibe cinco parámetros, uno para cada variable de instancia que inicializa. El método `dibujar` en las líneas 25 a 29 requiere un objeto `Graphics` y lo utiliza para dibujar la línea en el color apropiado y en las coordenadas correctas.

```
1  // Fig. 8.21: MiLinea.java
2  // Declaración de la clase MiLinea.
3  import java.awt.Color;
4  import java.awt.Graphics;
5
6  public class MiLinea
7  {
8     private int x1; // coordenada x del primer punto final
9     private int y1; // coordenada y del primer punto final
10    private int x2; // coordenada x del segundo punto final
```

Figura 8.21 | La clase `MiLinea` representa a una línea. (Parte 1 de 2).

```
   private int y2; // coordenada y del segundo punto final
   private Color miColor; // el color de esta figura

   // constructor con valores de entrada
   public MiLinea( int x1, int y1, int x2, int y2, Color color )
   {
      this.x1 = x1; // establece la coordenada x del primer punto final
      this.y1 = y1; // establece la coordenada y del primer punto final
      this.x2 = x2; // establece la coordenada x del segundo punto final
      this.y2 = y2; // establece la coordenada y del segundo punto final
      miColor = color; // establece el color
   } // fin del constructor de MiLinea

   // Dibuja la línea en el color específico
   public void dibujar( Graphics g )
   {
      g.setColor( miColor );
      g.drawLine( x1, y1, x2, y2 );
   } // fin del método dibujar
} // fin de la clase MiLinea
```

Figura 8.21 | La clase `MiLinea` representa a una línea. (Parte 2 de 2).

En la figura 8.22, declaramos la clase `PanelDibujo`, que generará objetos aleatorios de la clase `MiLinea`. La línea 12 declara un arreglo `MiLinea` para almacenar las líneas a dibujar. Dentro del constructor (líneas 15 a 37), la línea 17 establece el color de fondo a `Color.WHITE`. La línea 19 crea el arreglo con una longitud aleatoria entre 5 y 9. El ciclo en las líneas 22 a 36 crea un nuevo objeto `MiLinea` para cada elemento en el arreglo. Las líneas 25 a 28 generan coordenadas aleatorias para los puntos finales de cada línea, y las líneas 31 y 32 generan un color aleatorio para la línea. La línea 35 crea un nuevo objeto `MiLinea` con los valores generados al azar, y lo almacena en el arreglo.

```
1  // Fig. 8.22: PanelDibujo.java
2  // Programa que utiliza la clase MiLinea
3  // para dibujar líneas al azar.
4  import java.awt.Color;
5  import java.awt.Graphics;
6  import java.util.Random;
7  import javax.swing.JPanel;
8
9  public class PanelDibujo extends JPanel
10 {
11    private Random numerosAleatorios = new Random();
12    private MiLinea lineas[]; // arreglo de lineas
13
14    // constructor, crea un panel con figuras al azar
15    public PanelDibujo()
16    {
17       setBackground( Color.WHITE );
18
19       lineas = new MiLinea[ 5 + numerosAleatorios.nextInt( 5 ) ];
20
21       // crea lineas
22       for ( int cuenta = 0; cuenta < lineas.length; cuenta++ )
23       {
24          // genera coordenadas aleatorias
```

Figura 8.22 | Creación de objetos `MiLinea` al azar. (Parte 1 de 2).

```
25             int x1 = numerosAleatorios.nextInt( 300 );
26             int y1 = numerosAleatorios.nextInt( 300 );
27             int x2 = numerosAleatorios.nextInt( 300 );
28             int y2 = numerosAleatorios.nextInt( 300 );
29
30             // genera un color aleatorio
31             Color color = new Color( numerosAleatorios.nextInt( 256 ),
32                numerosAleatorios.nextInt( 256 ), numerosAleatorios.nextInt( 256 ) );
33
34             // agrega la línea a la lista de líneas a mostrar en pantalla
35             lineas[ cuenta ] = new MiLinea( x1, y1, x2, y2, color );
36          } // fin de for
37       } // fin del constructor de PanelDibujo
38
39       // para cada arreglo de figuras, dibuja las figuras individuales
40       public void paintComponent( Graphics g )
41       {
42          super.paintComponent( g );
43
44          // dibuja las líneas
45          for ( MiLinea linea : lineas )
46             linea.dibujar( g );
47       } // fin del método paintComponent
48    } // fin de la clase PanelDibujo
```

Figura 8.22 | Creación de objetos `MiLinea` al azar. (Parte 2 de 2).

El método `paintComponent` itera a través de los objetos `MiLinea` en el arreglo `lineas` usando una instrucción `for` mejorada (líneas 45 y 46). Cada iteración llama al método `dibujar` del objeto `MiLinea` actual, y le pasa el objeto `Graphics` para dibujar en el panel. La clase `PruebaDibujo` en la figura 8.23 establece una nueva ventana para mostrar nuestro dibujo. Como estableceremos las coordenadas para las líneas sólo una vez en el constructor, el dibujo no cambia si se hace una llamada a `paintComponent` para actualizar el dibujo en la pantalla.

Ejercicio del ejemplo práctico de GUI y gráficos

8.1 Extienda el programa de las figuras 8.21 a 8.23 para dibujar rectángulos y óvalos al azar. Cree las clases `MiRectangulo` y `MiOvalo`. Ambas deben incluir las coordenadas *x1*, *y1*, *x2*, *y2*, un color y una bandera `boolean` para determinar si la figura es rellena. Declare un constructor en cada clase con argumentos para inicializar todas las variables de instancia. Para ayudar a dibujar rectángulos y óvalos, cada clase debe proporcionar los métodos `obtenerXSupIzq`, `obtenerYSupIzq`, `obtenerAnchura` y `obtenerAltura`, que calculen la coordenada *x* superior izquierda, la coordenada *y* superior izquierda, la anchura y la altura, respectivamente. La coordenada *x* superior izquierda es el más pequeño de los dos valores de coordenada *x*, la coordenada *y* superior izquierda es el más pequeño de los dos valores de coordenada *y*, la anchura es el valor absoluto de la diferencia entre los dos valores de coordenada *x*, y la altura es el valor absoluto de la diferencia entre los dos valores de coordenada *y*.

La clase `PanelDibujo`, que extiende a `JPanel` y se encarga de la creación de las figuras, debe declarar tres arreglos, uno para cada tipo de figura. La longitud de cada arreglo debe ser un número aleatorio entre 1 y 5. El constructor de la clase `PanelDibujo` debe llenar cada uno de los arreglos con figuras de posición, tamaño, color y relleno aleatorios.

Además, modifique las tres clases de figuras para incluir lo siguiente:

a) Un constructor sin argumentos que establezca todas las coordenadas de la figura a 0, el color de la figura a `Color.BLACK` y la propiedad de relleno a `false` (sólo en `MiRectangulo` y `MiOvalo`).

b) Métodos *establecer* para las variables de instancia en cada clase. Los métodos para establecer el valor de una coordenada deben verificar que el argumento sea mayor o igual a cero, antes de establecer la coordenada; si no es así, deben establecer la coordenada a cero. El constructor debe llamar a los métodos *establecer*, en vez de inicializar las variables locales directamente.

c) Métodos *obtener* para las variables de instancia en cada clase. El método `dibujar` debe hacer referencia a las coordenadas mediante los métodos *obtener*, en vez de acceder a ellas directamente.

```
// Fig. 8.23: PruebaDibujo.java
// Aplicación de prueba para mostrar un PanelDibujo en pantalla.
import javax.swing.JFrame;

public class PruebaDibujo
{
   public static void main( String args[] )
   {
      PanelDibujo panel = new PanelDibujo();
      JFrame aplicacion = new JFrame();

      aplicacion.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
      aplicacion.add( panel );
      aplicacion.setSize( 300, 300 );
      aplicacion.setVisible( true );
   } // fin de main
} // fin de la clase PruebaDibujo
```



Figura 8.23 | Creación de un objeto `JFrame` para mostrar `PanelDibujo`.

8.19 (Opcional) Ejemplo práctico de Ingeniería de Software: inicio de la programación de las clases del sistema ATM

En las secciones del Ejemplo práctico de Ingeniería de Software de los capítulos 1 al 7, hablamos de los fundamentos de la orientación a objetos y desarrollamos un diseño orientado a objetos para nuestro sistema ATM. Anteriormente en este capítulo, vimos muchos de los detalles de programación con clases. Ahora empezaremos a implementar nuestro diseño orientado a objetos en Java. Al final de esta sección, le mostraremos cómo convertir los diagramas de clases en código de Java. En la sección final del Ejemplo práctico de Ingeniería de Software (sección 10.9), modificaremos el código para incorporar el concepto orientado a objetos de herencia. En el apéndice M presentamos la implementación completa del código en Java.

Visibilidad

Ahora aplicaremos modificadores de acceso a los miembros de nuestras clases. En el capítulo 3 presentamos los modificadores de acceso `public` y `private`. Los modificadores de acceso determinan la **visibilidad**, o accesibilidad, de los atributos y métodos de un objeto para otros objetos. Antes de empezar a implementar nuestro diseño, debemos considerar cuáles atributos y métodos de nuestras clases deben ser `public` y cuáles deben ser `private`.

En el capítulo 3 observamos que, por lo general, los atributos deben ser `private`, y que los métodos invocados por los clientes de una clase dada deben ser `public`. Sin embargo, los métodos que se llaman sólo por otros métodos de la clase como "métodos utilitarios" deben ser `private`. UML emplea **marcadores de visibilidad** para

modelar la visibilidad de los atributos y las operaciones. La visibilidad pública se indica mediante la colocación de un signo más (+) antes de una operación o atributo, mientras que un signo menos (-) indica una visibilidad privada. La figura 8.24 muestra nuestro diagrama de clases actualizado, en el cual se incluyen los marcadores de visibilidad. [*Nota:* no incluimos parámetros de operación en la figura 8.24; esto es perfectamente normal. Agregar los marcadores de visibilidad no afecta a los parámetros que ya están modelados en los diagramas de clases de las figuras 6.22 a 6.25].

Navegabilidad

Antes de empezar a implementar nuestro diseño en Java, presentaremos una notación adicional de UML. El diagrama de clases de la figura 8.25 refina aún más las relaciones entre las clases del sistema ATM, al agregar flechas de navegabilidad a las líneas de asociación. Las **flechas de navegabilidad** (representadas como flechas con puntas delgadas en el diagrama de clases) indican en qué dirección puede recorrerse una asociación. Al implementar un sistema diseñado mediante el uso de UML, los programadores utilizan flechas de navegabilidad para ayudar a determinar cuáles objetos necesitan referencias a otros objetos. Por ejemplo, la flecha de navegabilidad que apunta de la clase `ATM` a la clase `BaseDatosBanco` indica que podemos navegar de una a la otra, con lo cual se permite a la clase `ATM` invocar a las operaciones de `BaseDatosBanco`. No obstante, como la figura 8.25 no contiene una flecha de navegabilidad que apunte de la clase `BaseDatosBanco` a la clase `ATM`, la clase `BaseDatosBanco` no puede acceder a las operaciones de la clase `ATM`. Observe que las asociaciones en un diagrama de clases que tienen flechas de navegabilidad en ambos extremos, o que no tienen ninguna flecha, indican una **navegabilidad bidireccional**: la navegación puede proceder en cualquier dirección a lo largo de la asociación.

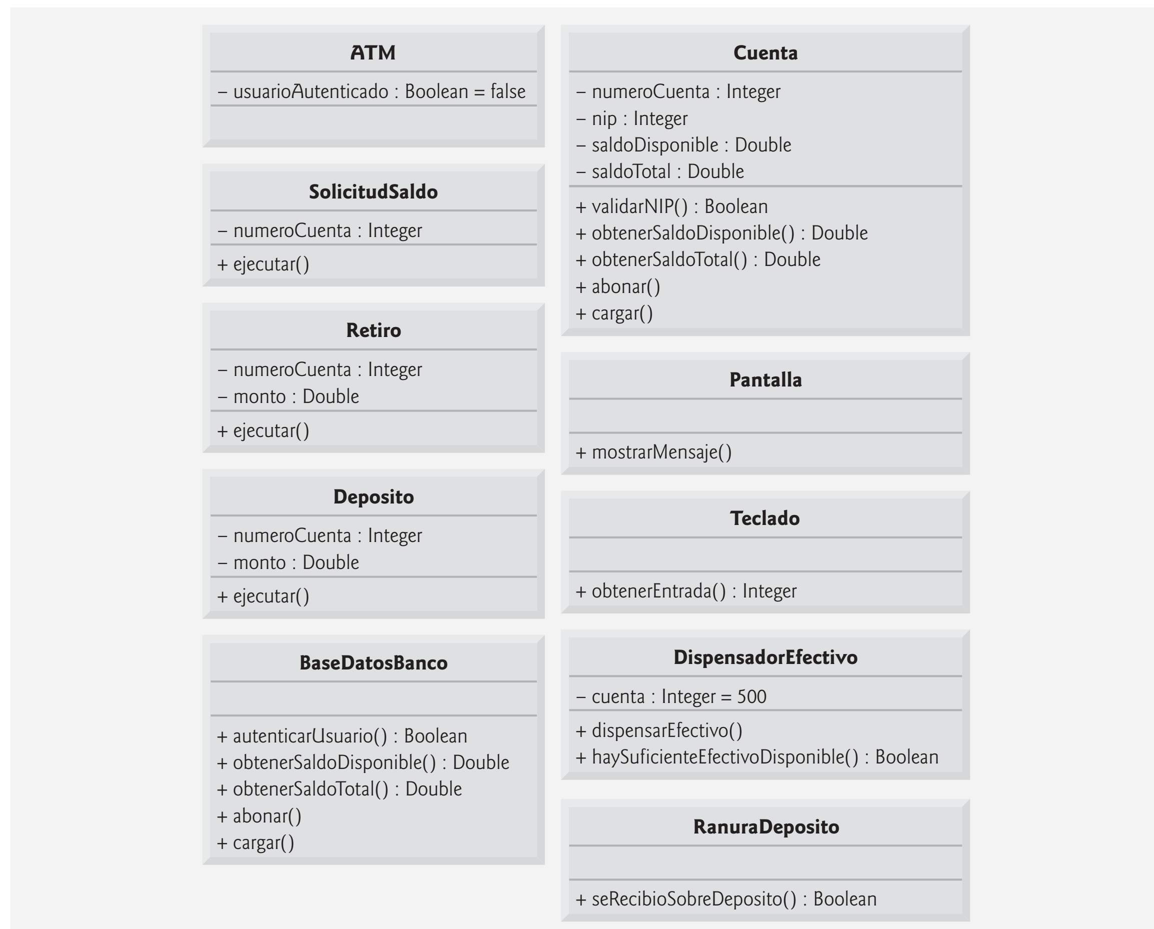


Figura 8.24 | Diagrama de clases con marcadores de visibilidad.

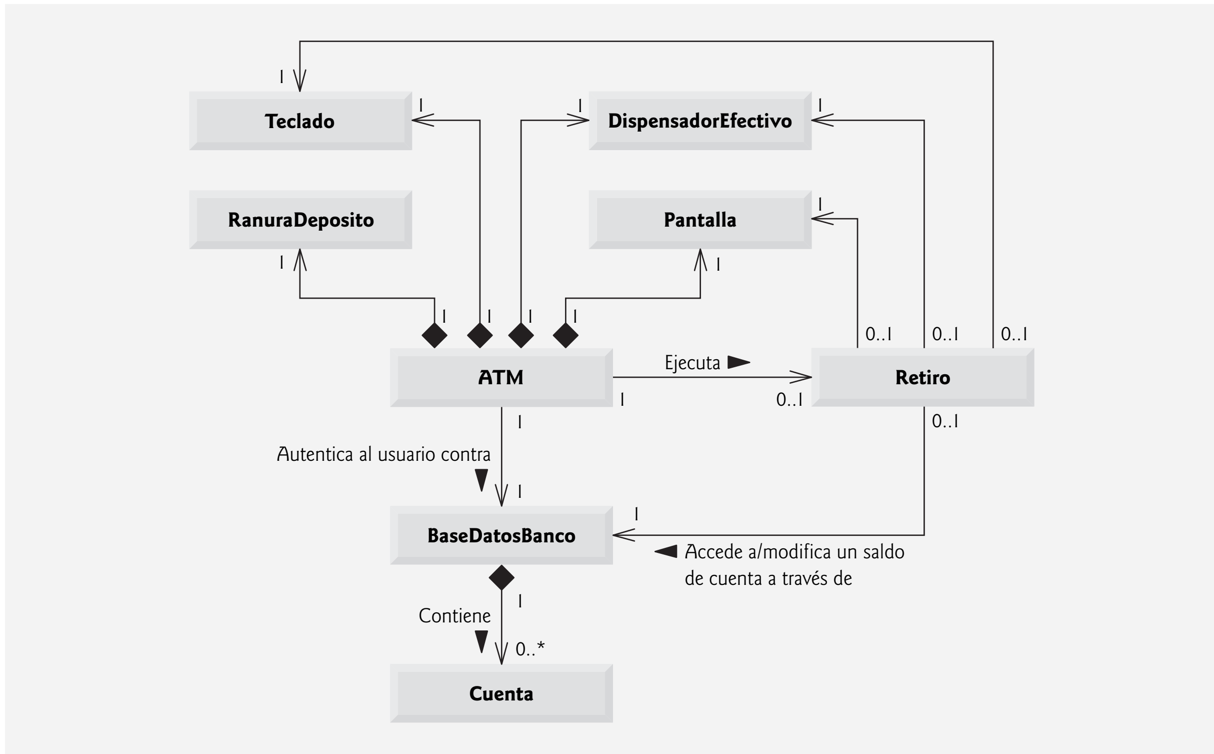


Figura 8.25 | Diagrama de clases con flechas de navegabilidad.

Al igual que el diagrama de clases de la figura 3.24, el de la figura 8.25 omite las clases `SolicitudSaldo` y `Deposito` para simplificarlo. La navegabilidad de las asociaciones en las que participan estas dos clases se asemeja mucho a la navegabilidad de las asociaciones de la clase `Retiro`. En la sección 3.10 vimos que `SolicitudSaldo` tiene una asociación con la clase `Pantalla`. Podemos navegar de la clase `SolicitudSaldo` a la clase `Pantalla` a lo largo de esta asociación, pero no podemos navegar de la clase `Pantalla` a la clase `SolicitudSaldo`. Por ende, si modeláramos la clase `SolicitudSaldo` en la figura 8.25, colocaríamos una flecha de navegabilidad en el extremo de la clase `Pantalla` de esta asociación. Recuerde también que la clase `Deposito` se asocia con las clases `Pantalla`, `Teclado` y `RanuraDeposito`. Podemos navegar de la clase `Deposito` a cada una de estas clases, pero no al revés. Por lo tanto, podríamos colocar flechas de navegabilidad en los extremos de las clases `Pantalla`, `Teclado` y `RanuraDeposito` de estas asociaciones. [*Nota:* modelaremos estas clases y asociaciones adicionales en nuestro diagrama de clases final en la sección 10.9, una vez que hayamos simplificado la estructura de nuestro sistema, al incorporar el concepto orientado a objetos de la herencia].

Implementación del sistema ATM a partir de su diseño de UML

Ahora estamos listos para empezar a implementar el sistema ATM. Primero convertiremos las clases de los diagramas de las figuras 8.24 y 8.25 en código de Java. Este código representará el "esqueleto" del sistema. En el capítulo 10 modificaremos el código para incorporar el concepto orientado a objetos de la herencia.

Como ejemplo, empezaremos a desarrollar el código a partir de nuestro diseño de la clase `Retiro` en la figura 8.24. Utilizaremos esta figura para determinar los atributos y operaciones de la clase. Usaremos el modelo de UML en la figura 8.25 para determinar las asociaciones entre las clases. Seguiremos estos cuatro lineamientos para cada clase:

1. Use el nombre que se localiza en el primer compartimiento para declarar la clase como `public`, con un constructor sin parámetros vacío. Incluimos este constructor tan sólo como un receptáculo para recordarnos que la mayoría de las clases necesitarán en definitiva constructores. En el apéndice M, en donde completamos una versión funcional de esta clase, agregaremos todos los argumentos y el código necesa-

rios al cuerpo del constructor. Por ejemplo, la clase `Retiro` produce el código de la figura 8.26. [*Nota:* si encontramos que las variables de instancia de la clase sólo requieren la inicialización predeterminada, eliminaremos el constructor sin parámetros vacío, ya que es innecesario].

2. Use los atributos que se localizan en el segundo compartimiento para declarar las variables de instancia. Por ejemplo, los atributos `private numeroCuenta` y `monto` de la clase `Retiro` producen el código de la figura 8.27. [*Nota:* el constructor de la versión funcional completa de esta clase asignará valores a estos atributos].

3. Use las asociaciones descritas en el diagrama de clases para declarar las referencias a otros objetos. Por ejemplo, de acuerdo con la figura 8.25, `Retiro` puede acceder a un objeto de la clase `Pantalla`, a un objeto de la clase `Teclado`, a un objeto de la clase `DispensadorEfectivo` y a un objeto de la clase `BaseDatosBanco`. Esto produce el código de la figura 8.28. [*Nota:* el constructor de la versión funcional completa de esta clase inicializará estas variables de instancia con referencias a objetos reales].

```java
// La clase Retiro representa una transacción de retiro del ATM
public class Retiro
{
   // constructor sin argumentos
   public Retiro()
   {
   } // fin del constructor de Retiro sin argumentos
} // fin de la clase Retiro
```

Figura 8.26 | Código de Java para la clase `Retiro`, con base en las figuras 8.24 y 8.25.

```java
// La clase Retiro representa una transacción de retiro del ATM
public class Retiro
{
   // atributos
   private int numeroCuenta; // cuenta de la que se van a retirar los fondos
   private double monto; // monto que se va a retirar de la cuenta

   // constructor sin argumentos
   public Retiro()
   {
   } // fin del constructor de Retiro sin argumentos
} // fin de la clase Retiro
```

Figura 8.27 | Código de Java para la clase `Retiro`, con base en las figuras 8.24 y 8.25.

```java
// La clase Retiro representa una transacción de retiro del ATM
public class Retiro
{
   // atributos
   private int numeroCuenta; // cuenta de la que se retirarán los fondos
   private double monto; // monto a retirar

   // referencias a los objetos asociados
   private Pantalla pantalla; // pantalla del ATM
   private Teclado teclado; // teclado del ATM
   private DispensadorEfectivo dispensadorEfectivo; // dispensador de efectivo del ATM
```

Figura 8.28 | Código de Java para la clase `Retiro`, con base en las figuras 8.24 y 8.25. (Parte 1 de 2).

```
   private BaseDatosBanco baseDatosBanco; // base de datos de información de las
   cuentas

   // constructor sin argumentos
   public Retiro()
   {
   } // fin del constructor de Retiro sin argumentos
} // fin de la clase Retiro
```

Figura 8.28 | Código de Java para la clase `Retiro`, con base en las figuras 8.24 y 8.25. (Parte 2 de 2).

4. Use las operaciones que se localizan en el tercer compartimiento de la figura 8.24 para declarar las armazones de los métodos. Si todavía no hemos especificado un tipo de valor de retorno para una operación, declaramos el método con el tipo de retorno `void`. Consulte los diagramas de clases de las figuras 6.22 a 6.25 para declarar cualquier parámetro necesario. Por ejemplo, al agregar la operación `public ejecutar` en la clase `Retiro`, que tiene una lista de parámetros vacía, se produce el código de la figura 8.29. [*Nota:* codificaremos los cuerpos de los métodos cuando implementemos el sistema ATM completo en el apéndice M].

```
// La clase Retiro representa una transacción de retiro del ATM
public class Retiro
{
   // atributos
   private int numeroCuenta; // cuenta de la que se van a retirar los fondos
   private double monto; // monto a retirar

   // referencias a los objetos asociados
   private Pantalla pantalla; // pantalla del ATM
   private Teclado teclado; // teclado del ATM
   private DispensadorEfectivo dispensadorEfectivo; // dispensador de efectivo del ATM
   private BaseDatosBanco baseDatosBanco; // base de datos de información de las cuentas

   // constructor sin argumentos
   public Retiro()
   {
   } // fin del constructor de Retiro sin argumentos

   // operaciones
   public void ejecutar()
   {
   } // fin del método ejecutar
} // fin de la clase Retiro
```

Figura 8.29 | Código de Java para la clase `Retiro`, con base en las figuras 8.24 y 8.25.

Esto concluye nuestra discusión sobre los fundamentos de la generación de clases a partir de diagramas de UML.

Ejercicios de autoevaluación del Ejemplo práctico de Ingeniería de Software

8.1 Indique si el siguiente enunciado es *verdadero* o *falso*, y si es *falso*, explique por qué: si un atributo de una clase se marca con un signo menos (-) en un diagrama de clases, el atributo no es directamente accesible fuera de la clase.

8.2 En la figura 8.25, la asociación entre los objetos `ATM` y `Pantalla` indica:
a) que podemos navegar de la `Pantalla` al `ATM`.
b) que podemos navegar del `ATM` a la `Pantalla`.

c) (a) y (b); la asociación es bidireccional.
d) Ninguna de las anteriores.

8.3 Escriba código de Java para empezar a implementar el diseño para la clase `Teclado`.

Respuestas a los ejercicios de autoevaluación del Ejemplo práctico de Ingeniería de Software

8.1 Verdadero. El signo menos (-) indica visibilidad privada.

8.2 b.

8.3 El diseño para la clase `Teclado` produce el código de la figura 8.30. Recuerde que la clase `Teclado` no tiene atributos en estos momentos, pero pueden volverse aparentes a medida que continuemos con la implementación. Observe además que, si fuéramos a diseñar un ATM real, el método `obtenerEntrada` tendría que interactuar con el hardware del teclado del ATM. En realidad recibiremos la entrada del teclado de una computadora personal, cuando escribamos el código de Java completo en el apéndice M.

```
// la clase Teclado representa el teclado de un ATM
public class Teclado
{
   // no se han especificado atributos todavía

   // constructor sin argumentos
   public Teclado()
   {
   } // fin del constructor de Teclado sin argumentos

   // operaciones
   public int obtenerEntrada()
   {
   } // fin del método obtenerEntrada
} // fin de la clase Teclado
```

Figura 8.30 | Código de Java para la clase `Teclado`, con base en las figuras 8.24 y 8.25.

8.20 Conclusión

En este capítulo presentamos conceptos adicionales de las clases. El ejemplo práctico de la clase `Tiempo` presentó una declaración de clase completa que consiste de datos `private`, constructores `public` sobrecargados para flexibilidad en la inicialización, métodos *establecer* y *obtener* para manipular los datos de la clase, y métodos que devuelven representaciones `String` de un objeto `Tiempo` en dos formatos distintos. Aprendió también que toda clase puede declarar un método `toString` que devuelva una representación `String` de un objeto de la clase, y que este método puede invocarse en forma implícita siempre que aparezca en el código un objeto de una clase, en donde se espera un `String`.

Aprendió que la referencia `this` se utiliza en forma implícita en los métodos no `static` de una clase para acceder a las variables de instancia de la clase y a otros métodos no `static`. Vio usos explícitos de la referencia `this` para acceder a los miembros de la clase (incluyendo los campos ocultos) y aprendió a utilizar la palabra clave `this` en un constructor para llamar a otro constructor de la clase.

Hablamos sobre las diferencias entre los constructores predeterminados que proporciona el compilador, y los constructores sin argumentos que proporciona el programador. Aprendió que una clase puede tener referencias a los objetos de otras clases como miembros; un concepto conocido como composición. Vio el tipo de clase `enum` y aprendió a usarlo para crear un conjunto de constantes para usarlas en un programa. Aprendió acerca de la capacidad de recolección de basura de Java y cómo reclama la memoria de los objetos que ya no se utilizan. Explicamos la motivación para los campos `static` en una clase, y le demostramos cómo declarar y utilizar campos y métodos `static` en sus propias clases. También aprendió a declarar e inicializar variables `final`.

Aprendió a empaquetar sus propias clases para reutilizarlas, y cómo importar esas clases en una aplicación. Le mostramos cómo crear una biblioteca de clases para reutilizar componentes, y cómo utilizar las clases de la

biblioteca en una aplicación. Por último, aprendió que los campos que se declaran sin un modificador de acceso reciben un acceso a nivel de paquete, de manera predeterminada. Vio la relación entre las clases en el mismo paquete, que permite a cada clase en un paquete acceder a los miembros con acceso a nivel de paquete de otras clases en ese mismo paquete.

En el siguiente capítulo aprenderá acerca de un aspecto importante de la programación orientada a objetos en Java: la herencia. En ese capítulo verá que todas las clases en Java se relacionan en forma directa o indirecta con la clase llamada `Object`. También empezará a comprender cómo las relaciones entre las clases le permiten crear aplicaciones más poderosas.

Resumen

Sección 8.2 Ejemplo práctico de la clase `Tiempo`

- Toda clase que usted declara representa un nuevo tipo en `Java`.
- Los métodos `public` de una clase se conocen también como los servicios `public` de la clase, o su interfaz `public`. El propósito principal de los estos métodos es presentar a los clientes de la clase una vista de los servicios que ésta proporciona. Los clientes de la clase no se necesitan preocupar por la forma en que ésta realiza sus tareas. Por esta razón, los miembros de clase `private` no son directamente accesibles para los clientes de la clase.
- Un objeto que contiene datos consistentes tiene valores de datos que siempre se mantienen dentro del rango.
- Un valor que se pasa a un método para modificar una variable de instancia es un valor correcto, si se encuentra dentro del rango permitido para la variable de instancia. Un valor correcto siempre consistente, pero un valor consistente no es correcto si un método recibe un valor fuera de rango, y lo establece en un valor consistente para mantener el objeto en un estado consistente.
- El método `static format` de la clase `String` es similar al método `System.out.printf`, excepto que `format` devuelve un objeto `String` con formato, en vez de mostrarlo en una ventana de comandos.
- Todos los objetos en Java tienen un método `toString`, que devuelve una representación `String` del objeto. El método `toString` se llama en forma implícita cuando aparece un objeto en el código en donde se requiere un `String`.

Sección 8.4 Referencias a los miembros del objeto actual mediante `this`

- Un método no `static` de un objeto utiliza en forma implícita la palabra clave `this` para hacer referencia a las variables de instancia del objeto, y a los demás métodos. La palabra clave `this` también se puede utilizar en forma explícita.
- El compilador produce un archivo separado con la extensión `.class` para cada clase compilada.
- Si un método contiene una variable local con el mismo nombre que uno de los campos de su clase, la variable local oculta el campo en el alcance del método. El método puede usar la referencia `this` para hacer referencia al campo oculto en forma explícita.

Sección 8.5 Ejemplo práctico de la clase `Tiempo`*: constructores sobrecargados*

- Los constructores sobrecargados permiten inicializar los objetos de una clase de varias formas distintas. El compilador diferencia a los constructores sobrecargados en base a sus firmas.

Sección 8.6 Constructores predeterminados y sin argumentos

- Toda clase debe tener por lo menos un constructor. Si no se proporciona uno, el compilador crea un constructor predeterminado, que inicializa las variables de instancia con los valores iniciales especificados en sus declaraciones, o con sus valores predeterminados.
- Si una clase declara constructores, el compilador no crea un constructor predeterminado. Para especificar la inicialización predeterminada para los objetos de una clase con varios constructores, el programador debe declarar un constructor sin argumentos.

Sección 8.7 Observaciones acerca de los métodos Establecer *y* Obtener

- Los métodos *establecer* se conocen comúnmente como métodos mutadores, ya que, por lo general, cambian un valor. Los métodos *obtener* se conocen comúnmente como métodos de acceso o de consulta. Un método predicado evalúa si una condición es verdadera o falsa.

Sección 8.8 Composición

- Una clase puede tener referencias a objetos de otras clases como miembros. A dicha capacidad se le conoce como composición, y algunas veces se le denomina relación *tiene un*.

Sección 8.9 Enumeraciones

- Todos los tipos `enum` son tipos por referencia. Un tipo `enum` se declara con una declaración `enum`, que es una lista separada por comas de constantes `enum`. La declaración puede incluir, de manera opcional, otros componentes de las clases tradicionales, como: constructores, campos y métodos.
- Los tipos `enum` son implícitamente `final`, ya que declaran constantes que no deben modificarse.
- Las constantes `enum` son implícitamente `static`.
- Cualquier intento por crear un objeto de un tipo `enum` con el operador `new` produce un error de compilación.
- Las constantes `enum` se pueden utilizar en cualquier parte en donde pueden usarse constantes, como en las etiquetas `case` de las instrucciones `switch` y para controlar las instrucciones `for` mejoradas.
- Cada constante `enum` en una declaración `enum` va seguida opcionalmente de argumentos que se pasan al constructor de la `enum`.
- Para cada `enum`, el compilador genera un método `static` llamado `values`, que devuelve un arreglo de las constantes de la `enum`, en el orden en el que se declararon.
- El método `static range` de `EnumSet` recibe dos parámetros: la primera constante `enum` en un rango y la última constante `enum` en un rango; y devuelve un objeto `EnumSet` que contiene todas las constantes entre estas dos constantes, inclusive.

Sección 8.10 Recolección de basura y el método `finalize`

- Toda clase en Java tiene los métodos de la clase `Object`, uno de los cuales es `finalize`.
- La Máquina Virtual de Java (JVM) realiza la recolección automática de basura para reclamar la memoria que ocupan los objetos que ya no se utilizan. Cuando ya no hay más referencias a un objeto, la JVM lo marca para la recolección de basura. La memoria para dicho objeto se puede reclamar cuando la JVM ejecuta su recolector de basura.
- El método `finalize` es invocado por el recolector de basura, justo antes de que reclame la memoria del objeto. El método `finalize` no recibe parámetros y tiene el tipo de retorno `void`.
- Tal vez el recolector de basura nunca se ejecute antes de que un programa termine. Por lo tanto, no queda claro si se hará una llamada al método `finalize` (o cuándo se hará).

Sección 8.11 Miembros de clase `static`

- Una variable `static` representa la información a nivel de clase que se comparte entre todos los objetos de la clase.
- Las variables `static` tienen alcance en toda la clase. Se puede tener acceso a los miembros `public static` de una clase a través de una referencia a cualquier objeto de la clase, o calificando el nombre del miembro con el nombre de la clase y un punto (`.`). El acceso a los miembros `private static` de una clase se obtiene sólo a través de los métodos de la clase.
- Los miembros de clase `static` existen aun cuando no existan objetos de la clase; están disponibles tan pronto como se carga la clase en memoria, en tiempo de ejecución. Para acceder a un miembro `private static` cuando no existen objetos de la clase, debe proporcionarse un método `public static`.
- El método `static gc` de la clase `System` indica que el recolector de basura debe realizar su mejor esfuerzo al tratar de reclamar objetos que sean candidatos para la recolección de basura.
- Un método que se declara como `static` no puede acceder a los miembros de clase que no son `static`, ya que un método `static` puede llamarse incluso aunque no se hayan creado instancias de objetos de la clase.
- La referencia `this` no puede utilizarse en un método `static`.

Sección 8.12 Declaración `static` import

- Una declaración `static` import permite a los programadores hacer referencia a los miembros `static` importados, sin tener que utilizar el nombre de la clase y un punto (`.`). Una declaración `static` import individual importa un miembro `static`, y una declaración `static` import sobre demanda importa a todos los miembros `static` de una clase.

Sección 8.13 Variables de instancia `final`

- En el contexto de una aplicación, el principio del menor privilegio establece que al código se le debe otorgar sólo el nivel de privilegio y de acceso que necesita para realizar su tarea designada.
- La palabra clave `final` especifica que una variable no puede modificarse; en otras palabras, es constante. Las constantes pueden inicializarse cuando se declaran, o por medio de cada uno de los constructores de una clase. Si una variable `final` no se inicializa, se produce un error de compilación.

Sección 8.14 Reutilización de software

- El software se construye a partir de componentes existentes, bien definidos, cuidadosamente probados, bien documentados, portables y con amplia disponibilidad. La reutilización de software agiliza el desarrollo de software poderoso, de alta calidad. El desarrollo rápido de aplicaciones (RAD) es de gran interés hoy en día.
- Ahora, los programadores de Java tienen miles de clases en la API a su disposición, para ayudarse a implementar programas en Java. Las clases de la API de Java permiten a los programadores de Java llevar nuevas aplicaciones al mercado con más rapidez, mediante el uso de componentes pre-existentes y probados.

Sección 8.15 Abstracción de datos y encapsulamiento

- El cliente de una clase se preocupa acerca de la funcionalidad que ésta ofrece, pero no acerca de cómo se implementa esta funcionalidad. A esto se le conoce como abstracción de datos. Aunque los programadores pueden conocer los detalles de la implementación de una clase, no deben escribir código que dependa de estos detalles. Esto nos permite reemplazar una clase con otra versión, sin afectar el resto del sistema.
- Un tipo de datos abstracto (ADT) consiste en una representación de datos y las operaciones que pueden realizarse con esos datos.

Sección 8.16 Ejemplo práctico de la clase `Tiempo`: creación de paquetes

- Cada clase en la API de Java pertenece a un paquete que contiene un grupo de clases relacionadas. Los paquetes ayudan a administrar la complejidad de los componentes de una aplicación, y facilitan la reutilización de software.
- Los paquetes proporcionan una convención para los nombres de clases únicos, que ayuda a evitar los conflictos de nombres de clases.
- Antes de poder importar una clase en varias aplicaciones, ésta debe colocarse en un paquete. Sólo puede haber una declaración `package` en cada archivo de código fuente de Java, y debe ir antes de todas las demás declaraciones e instrucciones en el archivo.
- Cada nombre de paquete debe empezar con el nombre de dominio de Internet del programador, en orden inverso. Una vez que se invierte el nombre de dominio, podemos elegir cualquier otro nombre que deseemos para nuestro paquete.
- Al compilar una clase en un paquete, la opción `-d` de línea de comandos de `javac` especifica en dónde se debe almacenar el `paquete`, y hace que el compilador cree los directorios del paquete, en caso de que no existan.
- El nombre del paquete forma parte del nombre completamente calificado de una clase. Esto ayuda a evitar los conflictos de nombres.
- Una declaración `import` de tipo simple especifica una clase a importar. Una declaración `import` de tipo por demanda sólo importa las clases que el programa utilice de un paquete específico.
- El compilador utiliza un cargador de clases para localizar las clases que necesita en la ruta de clases. La ruta de clases consiste en una lista de directorios o archivos de ficheros, cada uno separado por un separador de directorio.
- La ruta de clases para el compilador y la JVM se puede especificar proporcionando la opción `–classpath` al comando `javac` o `java`, o estableciendo la variable de entorno `CLASSPATH`. La ruta de clases para la JVM también se puede especificar mediante la opción `-cp` de línea de comandos. Si las clases deben cargarse del directorio actual, incluya un punto (`.`) en la ruta de clases.

Sección 8.17 Acceso a paquetes

- Si no se especifica un modificador de acceso para un método o variable al momento de su declaración en una clase, se considera que el método o variable tiene acceso a nivel de paquete.

Terminología

`-classpath`, argumento de línea de comandos para `javac`
`-d`, argumento de línea de comandos para `javac`
abstracción de datos
acceso a nivel de paquete
alcance de clase
archivo de ficheros
atributo
biblioteca de clases
cargador de clases
clase contenedora
`CLASSPATH`, variable de entorno
colisión de nombres
comportamiento
composición
comprobación de validez
conflicto de nombres
constructor predeterminado
constructor sin argumentos
constructores sobrecargados
declaración `import` de tipo por demanda
declaración `import` de tipo simple

declaración `static` import simple
desarrollo rápido de aplicaciones (RAD)
desbordamiento aritmético
`enum`, constante
`enum`, palabra clave
`EnumSet`, clase
`finalize`, método
`format`, método de la clase `String`
fuga de memoria
fuga de recursos
`gc`, método de la clase `System`
información a nivel de clase
lenguaje extensible
marcar un objeto para la recolección de basura
mecanismo de extensiones
método de consulta
método mutador
método predicado
modificador de acceso
nombre simple de una clase, campo o método
`package`, declaración
paquete opcional
pila
primero en entrar, primero en salir (PEPS)
principio de menor privilegio
`private`, modificador de acceso
`protected`, modificador de acceso
`public`, interfaz
`public`, modificador de acceso
`public`, servicio
`range`, método de `EnumSet`
recolector de basura
representación de datos
ruta de clases
separador de directorio
servicio de una clase
`static`, campo (variable de clase)
`static`, declaración import
`static`, declaración import por demanda
tareas de preparación para la terminación
`this`, palabra clave
tiene un, relación
tipo de datos abstracto (ADT)
último en entrar, primero en salir (UEPS)
`values`, método de una `enum`
variable constante
variable de clase
variable que no se puede modificar

Ejercicio de autoevaluación

8.1 Complete los siguientes enunciados:

a) Al compilar una clase en un paquete, la opción __________ de línea de comandos de `javac` especifica en dónde se debe almacenar el paquete, y hace que el compilador cree los directorios, en caso de que no existan.
b) El método `static` __________ de la clase `String` es similar al método `System.out.printf`, pero devuelve un objeto `String` con formato en vez de mostrar un objeto `String` en una ventana de comandos.
c) Si un método contiene una variable local con el mismo nombre que uno de los campos de su clase, la variable local __________ al campo en el alcance de ese método.
d) El recolector de basura llama al método __________ antes de reclamar la memoria de un objeto.
e) Una declaración __________ especifica una clase a importar.
f) Si una clase declara constructores, el compilador no creará un(a) __________.
g) El método __________ de un objeto se llama en forma implícita cuando aparece un objeto en el código, en donde se necesita un `String`.
h) A los métodos *establecer* se les llama comúnmente __________ o __________.
i) Un método __________ evalúa si una condición es verdadera o falsa.
j) Para cada `enum`, el compilador genera un método `static` llamado __________, que devuelve un arreglo de las constantes de la `enum` en el orden en el que se declararon.
k) A la composición se le conoce algunas veces como relación __________.
l) Una declaración __________ contiene una lista separada por comas de constantes.
m) Una variable __________ representa información a nivel de clase, que comparten todos los objetos de la clase.
n) Una declaración __________ importa un miembro `static`.
o) El __________ establece que al código se le debe otorgar sólo el nivel de privilegio y de acceso que necesita para realizar su tarea designada.
p) La palabra clave __________ especifica que una variable no se puede modificar.
q) Un(a) __________ consiste en una representación de datos y las operaciones que pueden realizarse sobre esos datos.

r) Sólo puede haber un(a) __________ en un archivo de código fuente de Java, y debe ir antes de todas las demás declaraciones e instrucciones en el archivo.
s) Un(a) declaración __________ sólo importa las clases que utiliza el programa de un paquete específico.
t) El compilador utiliza un(a) __________ para localizar las clases que necesita en la ruta de clases.
u) La ruta de clases para el compilador y la JVM se puede especificar mediante la opción __________ para el comando `javac` o `java`, o estableciendo la variable de entorno __________.
v) A los métodos *establecer* se les conoce comúnmente como __________, ya que, por lo general, modifican un valor.
w) Un(a) __________ importa a todos los miembros `static` de una clase.
x) Los métodos `public` de una clase se conocen también como los __________ o __________ de la clase.
y) El método `static` __________ de la clase `System` indica que el recolector de basura debe realizar su mejor esfuerzo para tratar de reclamar los objetos que sean candidatos para la recolección de basura.
z) Un objeto que contiene __________ tiene valores de datos que siempre se mantienen dentro del rango.

Respuestas a los ejercicios de autoevaluación

8.1 a) `-d`. b) `format`. c) oculta. d) `finalize`. e) import de tipo simple. f) constructor predeterminado. g) `toString`. h) métodos de acceso, métodos de consulta. i) predicado. j) `values`. k) *tiene un*. l) `enum`. m) `static`. n) `static import` de tipo simple. o) principio de menor privilegio. p) `final`. q) tipo de datos abstracto (ADT). r) declaración `package`. s) `import` tipo sobre demanda. t) cargador de clases. u) `-classpath`, `CLASSPATH`. v) métodos mutadores. w) declaración `static import` sobre demanda. x) servicios `public`, interfaz `public`. y) `gc`. z) datos consistentes.

Ejercicios

8.2 Explique la noción del acceso a nivel de paquete en Java. Explique los aspectos negativos del acceso a nivel de paquete.

8.3 ¿Qué ocurre cuando un tipo de valor de retorno, incluso `void`, se especifica para un constructor?

8.4 *(Clase Rectangulo)* Cree una clase llamada `Rectangulo`. La clase debe tener los atributos `longitud` y `anchura`, cada uno con un valor predeterminado de 1. Debe tener métodos para calcular el `perimetro` y el `area` del rectángulo. Debe tener métodos *establecer* y *obtener* para `longitud` y `anchura`. Los métodos *establecer* deben verificar que `longitud` y `anchura` sean números de punto flotante mayores de 0.0, y menores de 20.0. Escriba un programa para probar la clase `Rectangulo`.

8.5 *(Modificación de la representación de datos interna de una clase)* Sería perfectamente razonable para la clase `Tiempo2` de la figura 8.5 representar la hora internamente como el número de segundos transcurridos desde medianoche, en vez de usar los tres valores enteros `hora`, `minuto` y `segundo`. Los clientes podrían usar los mismos métodos `public` y obtener los mismos resultados. Modifique la clase `Tiempo2` de la figura 8.5 para implementar un objeto `Tiempo2` como el número de segundos transcurridos desde medianoche, y mostrar que no hay cambios visibles para los clientes de la clase.

8.6 *(Clase cuenta de ahorros)* Cree una clase llamada `CuentaDeAhorros`. Use una variable `static` llamada `tasaInteresAnual` para almacenar la tasa de interés anual para todos los cuentahabientes. Cada objeto de la clase debe contener una variable de instancia `private` llamada `saldoAhorros`, que indique la cantidad que el ahorrador tiene actualmente en depósito. Proporcione el método `calcularInteresMensual` para calcular el interés mensual, multiplicando el `saldoAhorros` por la `tasaInteresAnual` dividida entre 12; este interés debe sumarse al `saldoAhorros`. Proporcione un método `static` llamado `modificarTasaInteres` para establecer la `tasaInteresAnual` en un nuevo valor. Escriba un programa para probar la clase `CuentaDeAhorros`. Cree dos instancias de objetos `CuentaDeAhorros`, `ahorrador1` y `ahorrador2`, con saldos de $2000.00 y $3000.00, respectivamente. Establezca la `tasaInteresAnual` en 4%, después calcule el interés mensual e imprima los nuevos saldos para ambos ahorradores. Luego establezca la `tasaInteresAnual` en 5%, calcule el interés del siguiente mes e imprima los nuevos saldos para ambos ahorradores.

8.7 *(Mejora a la clase* `Tiempo2`*)* Modifique la clase `Tiempo2` de la figura 8.5 para incluir un método `tictac`, que incremente el tiempo almacenado en un objeto `Tiempo2` por un segundo. Proporcione el método `incrementarMinuto` para incrementar el minuto, y el método `incrementarHora` para incrementar la hora. El objeto `Tiempo2` debe permanecer siempre en un estado consistente. Escriba un programa para probar los métodos `tictac`, `incrementarMinuto` y `incrementarHora`, para asegurarse que funcionen correctamente. Asegúrese de probar los siguientes casos:

a) Incrementar el minuto, de manera que cambie al siguiente minuto.
b) Incrementar la hora, de manera que cambie a la siguiente hora.
c) Incrementar el tiempo de manera que cambie al siguiente día (por ejemplo, de 11:59:59 PM a 12:00:00 AM).

8.8 *(Mejora a la clase Fecha)* Modifique la clase `Fecha` de la figura 8.7 para realizar la comprobación de errores en los valores inicializadores para las variables de instancia `mes`, `dia` y `anio` (la versión actual sólo valida el mes y el día). Proporcione un método llamado `siguienteDia` para incrementar el día en uno. El objeto `Fecha` siempre deberá permanecer en un estado consistente. Escriba un programa que evalúe el método `siguienteDia` en un ciclo que imprima la fecha durante cada iteración del ciclo, para mostrar que el método `siguienteDia` funciona correctamente. Pruebe los siguientes casos:

a) Incrementar la fecha de manera que cambie al siguiente mes.
b) Incrementar la fecha de manera que cambie al siguiente año.

8.9 *(Devolver indicadores de errores de los métodos)* Modifique los métodos *establecer* en la clase `Tiempo2` de la figura 8.5 para devolver valores de error apropiados si se hace un intento por establecer una de las variables de instancia `hora`, `minuto` o `segundo` de un objeto de la clase `Tiempo`, en un valor inválido. [*Sugerencia:* use tipos de valores de retorno `boolean` en cada método]. Escriba un programa que pruebe estos nuevos métodos *establecer* y que imprima mensajes de error cuando se reciban valores incorrectos.

8.10 Vuelva a escribir la figura 8.14, de manera que utilice una declaración `import` separada para cada miembro `static` de la clase `Math` que se utilice en el ejemplo.

8.11 Escriba un tipo `enum` llamado `LuzSemaforo`, cuyas constantes (`ROJO`, `VERDE`, `AMARILLO`) reciban un parámetro: la duración de la luz. Escriba un programa para probar la `enum LuzSemaforo`, de manera que muestre las constantes de la `enum` y sus duraciones.

8.12 *(Números complejos)* Cree una clase llamada `Complejo` para realizar operaciones aritméticas con números complejos. Estos números tienen la forma

parteReal + *parteImaginaria* * *i*

en donde *i* es

$\sqrt{-1}$

Escriba un programa para probar su clase. Use variables de punto flotante para representar los datos `private` de la clase. Proporcione un constructor que permita que un objeto de esta clase se inicialice al declararse. Proporcione un constructor sin argumentos con valores predeterminados, en caso de que no se proporcionen inicializadores. Proporcione métodos `public` que realicen las siguientes operaciones:

a) Sumar dos números `Complejo`: las partes reales se suman entre sí y las partes imaginarias también.
b) Restar dos números `Complejo`: la parte real del operando derecho se resta de la parte real del operando izquierdo, y la parte imaginaria del operando derecho se resta de la parte imaginaria del operando izquierdo.
c) Imprimir números `Complejo` en la forma `(a, b)`, en donde `a` es la parte real y `b` es la imaginaria.

8.13 *(Clase Fecha y Tiempo)* Cree una clase llamada `FechaYTiempo`, que combine la clase `Tiempo2` modificada del ejercicio 8.7 y la clase `Fecha` modificada del ejercicio 8.8. Modifique el método `incrementarHora` para llamar al método `siguienteDia` si el tiempo se incrementa hasta el siguiente día. Modifique los métodos `aStringEstandar` y `aStringUniversal` para imprimir la fecha, junto con la hora. Escriba un programa para evaluar la nueva clase `FechaYTiempo`. En específico, pruebe incrementando la hora para que cambie al siguiente día.

8.14 *(Clase Rectangulo mejorada)* Cree una clase `Rectangulo` más sofisticada que la que creó en el ejercicio 8.4. Esta clase debe guardar solamente las coordenadas Cartesianas de las cuatro esquinas del rectángulo. El constructor debe llamar a un método *establecer* que acepte cuatro conjuntos de coordenadas y verifique que cada una de éstas se encuentre en el primer cuadrante, en donde ninguna coordenada *x* o *y* debe ser mayor de 20.0. El método *establecer* debe también verificar que las coordenadas proporcionadas especifiquen un rectángulo. Proporcione métodos para calcular la `longitud`, `anchura`, `perimetro` y `area`. La longitud será la mayor de las dos dimensiones. Incluya un método predicado llamado `esCuadrado`, el cual determine si el rectángulo es un cuadrado. Escriba un programa para probar la clase `Rectangulo`.

8.15 *(Conjunto de enteros)* Cree la clase `ConjuntoEnteros`. Cada objeto `ConjuntoEnteros` puede almacenar enteros en el rango de 0 a 100. El conjunto se representa mediante un arreglo de valores `boolean`. El elemento del arreglo `a[i]` es `true` si el entero *i* se encuentra en el conjunto. El elemento del arreglo `a[j]` es `false` si el entero *j* no se encuentra

dentro del conjunto. El constructor sin argumentos inicializa el arreglo de Java con el "conjunto vacío" (es decir, un conjunto cuya representación de arreglo contiene sólo valores `false`).

Proporcione los siguientes métodos: el método `union` debe crear un tercer conjunto que sea la unión teórica de conjuntos para los dos conjuntos existentes (es decir, un elemento del tercer arreglo se establece en `true` si ese elemento es `true` en cualquiera o en ambos de los conjuntos existentes; en caso contrario, el elemento del tercer conjunto se establece en `false`). El método `interseccion` debe crear un tercer conjunto que sea la intersección teórica de conjuntos para los dos conjuntos existentes (es decir, un elemento del arreglo del tercer conjunto se establece en `false` si ese elemento es `false` en uno o ambos de los conjuntos existentes; en caso contrario, el elemento del tercer conjunto se establece en `true`). El método `insertarElemento` debe insertar un nuevo entero *k* en un conjunto (estableciendo `a[k]` en `true`). El método `eliminarElemento` debe eliminar el entero *m* (estableciendo `a[m]` en `false`). El método `aStringConjunto` debe devolver una cadena que contenga un conjunto como una lista de números separados por espacios. Incluya sólo aquellos elementos que estén presentes en el conjunto. Use `---` para representar un conjunto vacío. El método `esIgualA` debe determinar si dos conjuntos son iguales. Escriba un programa para probar la clase `ConjuntoEnteros`. Cree instancias de varios objetos `ConjuntoEnteros`. Pruebe que todos sus métodos funcionen correctamente.

8.16 *(Clase Fecha)* Cree la clase `Fecha` con las siguientes capacidades:

a) Imprimir la fecha en varios formatos, como

```
MM/DD/AAAA
Junio 15, 1992
DDD AAAA
```

b) Usar constructores sobrecargados para crear objetos `Fecha` inicializados con fechas de los formatos en la parte (a). En el primer caso, el constructor debe recibir tres valores enteros. En el segundo caso, debe recibir un objeto `String` y dos valores enteros. En el tercer caso debe recibir dos valores enteros, el primero de los cuales representa el número de día en el año. [*Sugerencia:* para convertir la representación de cadena del mes a un valor numérico, compare las cadenas usando el método `equals`. Por ejemplo, si `s1` y `s2` son cadenas, la llamada al método `s1.equals( s2 )` devuelve `true` si las cadenas son idénticas y devuelve `false` en cualquier otro caso].

8.17 *(Números racionales)* Cree una clase llamada `Racional` para realizar operaciones aritméticas con fracciones. Escriba un programa para probar su clase. Use variables enteras para representar las variables de instancia `private` de la clase: el `numerador` y el `denominador`. Proporcione un constructor que permita a un objeto de esta clase inicializarse al ser declarado. El constructor debe almacenar la fracción en forma reducida. La fracción

```
2/4
```

es equivalente a `1/2` y debe guardarse en el objeto como 1 en el `numerador` y 2 en el `denominador`. Proporcione un constructor sin argumentos con valores predeterminados, en caso de que no se proporcionen inicializadores. Proporcione métodos `public` que realicen cada una de las siguientes operaciones:

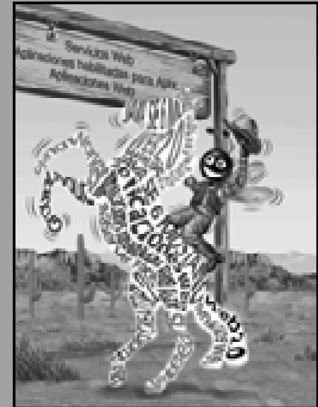
a) Sumar dos números `Racional`: el resultado de la suma debe almacenarse en forma reducida.
b) Restar dos números `Racional`: el resultado de la resta debe almacenarse en forma reducida.
c) Multiplicar dos números `Racional`: el resultado de la multiplicación debe almacenarse en forma reducida.
d) Dividir dos números `Racional`: el resultado de la división debe almacenarse en forma reducida.
e) Imprimir números `Racional` en la forma `a/b`, en donde `a` es el `numerador` y `b` es el `denominador`.
f) Imprimir números `Racional` en formato de punto flotante. (Considere proporcionar capacidades de formato, que permitan al usuario de la clase especificar el número de dígitos de precisión a la derecha del punto decimal).

8.18 *(Clase Entero Enorme)* Cree una clase llamada `EnteroEnorme` que utilice un arreglo de 40 elementos de dígitos, para guardar enteros de hasta 40 dígitos de longitud cada uno. Proporcione los métodos `entrada`, `salida`, `sumar` y `restar`. Para comparar objetos `EnteroEnorme`, proporcione los siguientes métodos: `esIgualA`, `noEsIgualA`, `esMayorQue`, `esMenorQue`, `esMayorOIgualA`, y `esMenorOIgualA`. Cada uno de estos métodos deberá ser un método predicado que devuelva `true` si la relación se aplica entre los dos objetos `EnteroEnorme`, y `false` si no se aplica. Proporcione un método predicado llamado `esCero`. Si desea hacer algo más, proporcione también los métodos `multiplicar`, `dividir` y `residuo`. [*Nota:* los valores `boolean` primitivos pueden imprimirse como la palabra "true" o la palabra "false", con el especificador de formato `%b`].

8.19 *(Tres en raya)* Cree una clase llamada `TresEnRaya` que le permita escribir un programa completo para jugar al "tres en raya" (o tres en línea). La clase debe contener un arreglo privado bidimensional de enteros, con un tamaño de 3 por 3. El constructor debe inicializar el tablero vacío con ceros. Permita dos jugadores humanos. Siempre que el primer jugador realice un movimiento, coloque un 1 en el cuadro especificado; coloque un 2 siempre que el segundo jugador realice un movimiento. Cada movimiento debe hacerse en un cuadro vacío. Después de cada movimiento, determine si el juego se ha ganado o si hay un empate. Si desea hacer algo más, modifique su programa de manera que la computadora realice los movimientos para uno de los jugadores. Además, permita que el jugador especifique si desea el primer o segundo turno. Si se siente todavía más motivado, desarrolle un programa que reproduzca un juego de Tres en raya tridimensional, en un tablero de 4 por 4 por 4 [*Nota:* ¡éste es un proyecto retador que podría requerir de muchas semanas de esfuerzo!].

9

Programación orientada a objetos: herencia



No digas que conoces a alguien por completo, hasta que tengas que dividir una herencia con él.

—Johann Kasper Lavater

Este método es para definirse como el número de la clase de todas las clases similares a la clase dada.

—Bertrand Russell

Es bueno heredar una biblioteca, pero es mejor coleccionar una.

—Augustine Birrell

Preserva la autoridad base de los libros de otros.

—William Shakespeare

OBJETIVOS

En este capítulo aprenderá a:

- Comprender cómo la herencia fomenta la reutilización de software.
- Entender qué son las superclases y las subclases.
- Utilizar la palabra clave `extends` para crear una clase que herede los atributos y comportamientos de otra clase.
- Comprender el uso del modificador de acceso `protected` para dar a los métodos de la subclase acceso a los miembros de la superclase.
- Utilizar los miembros de superclases mediante `super`.
- Comprender cómo se utilizan los constructores en las jerarquías de herencia.
- Conocer los métodos de la clase `Object`, la superclase directa o indirecta de todas las clases en Java.

Plan general

9.1 Introducción

En este capítulo continuamos nuestra discusión acerca de la programación orientada a objetos (POO), introduciendo una de sus características principales, la **herencia**, que es una forma de reutilización de software en la que se crea una nueva clase absorbiendo los miembros de una clase existente, y se mejoran con nuevas capacidades, o con modificaciones en las capacidades ya existentes. Con la herencia, los programadores ahorran tiempo durante el desarrollo, al reutilizar software probado y depurado de alta calidad. Esto también aumenta la probabilidad de que un sistema se implemente con efectividad.

Al crear una clase, en vez de declarar miembros completamente nuevos, el programador puede designar que la nueva clase herede los miembros de una clase existente. Esta clase existente se conoce como **superclase**, y la nueva clase se conoce como **subclase**. (El lenguaje de programación C++ se refieren a la superclase como la **clase base**, y a la subclase como **clase derivada**). Cada subclase puede convertirse en la superclase de futuras subclases.

Una subclase generalmente agrega sus propios campos y métodos. Por lo tanto, una subclase es más específica que su superclase y representa a un grupo más especializado de objetos. Generalmente, la subclase exhibe los comportamientos de su superclase junto con comportamientos adicionales específicos de esta subclase. Es por ello que a la herencia se le conoce algunas veces como **especialización**.

La **superclase directa** es la superclase a partir de la cual la subclase hereda en forma explícita. Una **superclase indirecta** es cualquier clase arriba de la superclase directa en la **jerarquía de clases**, la cual define las relaciones de herencia entre las clases. En Java, la jerarquía de clases empieza con la clase `Object` (en el paquete `java.lang`), a partir de la cual se **extienden** (o "heredan") *todas* las clases en Java, ya sea en forma directa o indirecta. La sección 9.7 lista los métodos de la clase `Object`, de la cual heredan todas las demás clases. En el caso de la **herencia simple**, una clase se deriva de una superclase directa. Java, a diferencia de C++, no soporta la herencia múltiple (que ocurre cuando una clase se deriva de más de una superclase directa). En el capítulo 10, Programación orientada a objetos: polimorfismo, explicaremos cómo los programadores en Java pueden usar las interfaces para obtener muchos de los beneficios de la herencia múltiple, evitando al mismo tiempo los problemas asociados.

La experiencia en la creación de sistemas de software nos indica que algunas cantidades considerables de código tratan con casos especiales, estrechamente relacionados. Cuando los programadores se preocupan con casos especiales, los detalles pueden oscurecer el panorama general. Con la programación orientada a objetos, los programadores se enfocan en los elementos comunes entre los objetos en el sistema, en vez de enfocarse en los casos especiales.

Es necesario hacer una diferencia entre la **relación "*es un*"** y la **relación "*tiene un*"**. La relación *"es un"* representa a la herencia. En este tipo de relación, un objeto de una subclase puede tratarse también como un objeto de

su superclase. Por ejemplo, un automóvil *es un* vehículo. En contraste, la relación *"tiene un"* identifica a la composición (vea el capítulo 8). En este tipo de relación, un objeto contiene referencias a objetos como miembros. Por ejemplo, un automóvil *tiene un* volante de dirección (y un objeto automóvil tiene una referencia a un objeto volante de dirección).

Las clases nuevas pueden heredar de las clases en las **bibliotecas de clases**. Las organizaciones desarrollan sus propias bibliotecas de clases y pueden aprovechar las que ya están disponibles en todo el mundo. Es probable que algún día, la mayoría de software nuevo se construya a partir de **componentes reutilizables estándar**, como sucede actualmente con la mayoría de los automóviles y del hardware de computadora. Esto facilitará el desarrollo de software más poderoso, abundante y económico.

9.2 Superclases y subclases

A menudo, un objeto de una clase *"es un"* objeto de otra clase también. Por ejemplo, en la geometría un rectángulo *es un* cuadrilátero (al igual que los cuadrados, los paralelogramos y los trapezoides). Por lo tanto, en Java puede decirse que la clase `Rectangulo` hereda de la clase `Cuadrilatero`. En este contexto, la clase `Cuadrilatero` es una superclase, y la clase `Rectangulo` es una subclase. Un rectángulo *es un* tipo específico de cuadrilátero, pero es incorrecto decir que todo cuadrilátero *es un* rectángulo; el cuadrilátero podría ser un paralelogramo o alguna otra figura. En la figura 9.1 se muestran varios ejemplos sencillos de superclases y subclases; observe que las superclases tienden a ser "más generales", y las subclases "más específicas".

Como todo objeto de una subclase *"es un"* objeto de su superclase, y como una superclase puede tener muchas subclases, el conjunto de objetos representados por una superclase generalmente es más grande que el conjunto de objetos representado por cualquiera de sus subclases. Por ejemplo, la superclase `Vehiculo` representa a todos los vehículos, incluyendo automóviles, camiones, barcos, bicicletas, etcétera. En contraste, la subclase `Auto` representa a un subconjunto más pequeño y específico de los vehículos.

Las relaciones de herencia forman estructuras jerárquicas en forma de árbol. Una superclase existe en una relación jerárquica con sus subclases. Cuando las clases participan en relaciones de herencia, se "afilian" con otras clases. Una clase se convierte ya sea en una superclase, proporcionando miembros a otras clases, o en una subclase, heredando sus miembros de otras clases. En algunos casos, una clase es tanto superclase como subclase.

Desarrollaremos una jerarquía de clases de ejemplo (figura 9.2), también conocida como **jerarquía de herencia**. Una comunidad universitaria tiene miles de miembros, compuestos por empleados, estudiantes y exalumnos. Los empleados pueden ser miembros del cuerpo docente o administrativo. Los miembros del cuerpo docente pueden ser administradores (como decanos o jefes de departamento) o maestros. Observe que la jerarquía podría contener muchas otras clases. Por ejemplo, los estudiantes pueden ser graduados o no graduados. Los no graduados pueden ser de primero, segundo, tercero o cuarto año.

Cada flecha en la jerarquía representa una relación *"es un"*. Por ejemplo, al seguir las flechas en esta jerarquía de clases podemos decir "un `Empleado` *es un* `MiembroDeLaComunidad`" y "un `Maestro` *es un* miembro `Docente`". `MiembroDeLaComunidad` es la superclase directa de `Empleado`, `Estudiante` y `Exalumno`, y es una superclase indirecta de todas las demás clases en el diagrama. Si comienza desde la parte inferior del diagrama, podrá seguir las flechas y aplicar la relación *es-un* hasta la superclase superior. Por ejemplo, un `Administrador` *es un* miembro `Docente`, *es un* `Empleado` y *es un* `MiembroDeLaComunidad`.

Ahora considere la jerarquía de herencia de `Figura` en la figura 9.3. Esta jerarquía empieza con la superclase `Figura`, la cual se extiende mediante las subclases `FiguraBidimensional` y `FiguraTridimensional`; las `Figu-`

Superclase	Subclases
`Estudiante`	`EstudianteGraduado`, `EstudianteNoGraduado`.
`Figura`	`Circulo`, `Triangulo`, `Rectangulo`.
`Prestamo`	`PrestamoAutomovil`, `PrestamoMejoraCasa`, `PrestamoHipotecario`.
`Empleado`	`Docente`, `Administrativo`.
`CuentaBancaria`	`CuentaDeCheques`, `CuentaDeAhorros`.

Figura 9.1 | Ejemplos de herencia.



Figura 9.2 | Jerarquía de herencia para objetos `MiembroDeLaComunidad` universitaria.

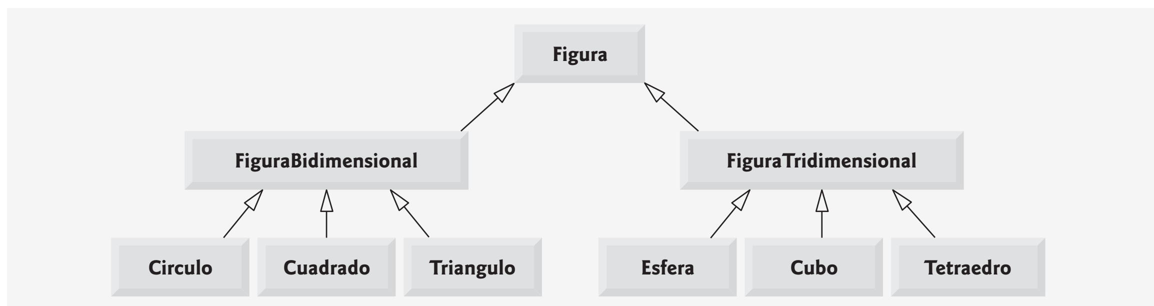


Figura 9.3 | Jerarquía de herencia para `Figuras`.

`ras` son del tipo `FiguraBidimensional` o `FiguraTridimensional`. El tercer nivel de esta jerarquía contiene algunos tipos más específicos de figuras tipo `FiguraBidimensional` y `FiguraTridimensional`. Al igual que en la figura 9.2, podemos seguir las flechas desde la parte inferior del diagrama, hasta la superclase de más arriba en esta jerarquía de clases, para identificar varias relaciones *es un*. Por ejemplo, un `Triangulo` *es un* objeto `FiguraBidimensional` y *es una* `Figura`, mientras que una `Esfera` *es una* `FiguraTridimensional` y *es una* `Figura`. Observe que esta jerarquía podría contener muchas otras clases. Por ejemplo, las elipses y los trapezoides son del tipo `FiguraBidimensional`.

No todas las relaciones de clases son una relación de herencia. En el capítulo 8 hablamos sobre la relación *tiene-un*, en la que las clases tienen miembros que hacen referencia a los objetos de otras clases. Tales relaciones crean clases mediante la composición de clases existentes. Por ejemplo, dadas las clases `Empleado`, `FechaDeNacimiento` y `NumeroTelefonico`, no es apropiado decir que un `Empleado` *es una* `FechaDeNacimiento` o que un `Empleado` *es un* `NumeroTelefonico`. Sin embargo, un `Empleado` *tiene una* `FechaDeNacimiento` y también *tiene un* `NumeroTelefonico`.

Es posible tratar a los objetos de superclases y a los objetos de subclases de manera similar; sus similitudes se expresan en los miembros de la superclase. Los objetos de todas las clases que extienden a una superclase común pueden tratarse como objetos de esa superclase (es decir, dichos objetos tienen una relación *"es un"* con la superclase). Sin embargo, los objetos de una superclase no pueden tratarse como objetos de sus subclases. Por ejemplo, todos los automóviles son vehículos pero no todos los vehículos son automóviles (los otros vehículos podrían ser camiones, aviones o bicicletas, por ejemplo). Más adelante en este capítulo y en el 10, Programación orientada a objetos: polimorfismo, consideraremos muchos ejemplos que aprovechan la relación *es un*.

Un problema con la herencia es que una subclase puede heredar métodos que no necesita, o que no debe tener. A pesar de que un método de superclase sea apropiado para una subclase, a menudo esa subclase requiere una versión personalizada del método. En dichos casos, la subclase puede **sobrescribir** (redefinir) el método de la superclase con una implementación apropiada, como veremos a menudo en los ejemplos de código de este capítulo.

9.3 Miembros protected

En el capítulo 8 hablamos sobre los modificadores de acceso `public` y `private`. Los miembros `public` de una clase son accesibles en cualquier parte en donde el programa tenga una referencia a un objeto de esa clase, o una de sus subclases. Los miembros `private` de una clase son accesibles sólo dentro de la misma clase. Los miembros `private` de una superclase no son heredados por sus subclases. En esta sección presentaremos el modificador de acceso `protected`. El uso del acceso `protected` ofrece un nivel intermedio de acceso entre `public` y `private`. Los miembros `protected` de una superclase pueden ser utilizados por los miembros de esa superclase, por los miembros de sus subclases y por los miembros de otras clases en el mismo paquete (los miembros `protected` también tienen acceso a nivel de paquete).

Todos los miembros `public` y `protected` de una superclase retienen su modificador de acceso original cuando se convierten en miembros de la subclase (por ejemplo, los miembros `public` de la superclase se convierten en miembros `public` de la subclase, y los miembros `protected` de la superclase se convierten en miembros `protected` de la subclase).

Los métodos de una subclase pueden referirse a los miembros `public` y `protected` que se hereden de la superclase con sólo utilizar los nombres de los miembros. Cuando un método de la subclase sobrescribe al método de la superclase, éste último puede utilizarse desde la subclase si se antepone a su nombre la palabra clave `super` y un punto (**.**). En la sección 9.4 hablaremos sobre el acceso a los miembros sobrescritos de la superclase.



Observación de ingeniería de software 9.1

Los métodos de una subclase no pueden tener acceso directo a los miembros `private` *de su superclase. Una subclase puede modificar el estado de las variables de instancia* `private` *de la superclase sólo a través de los métodos que no sean* `private`*, que se proporcionan en la superclase y son heredados por la subclase.*



Observación de ingeniería de software 9.2

Declarar variables de instancia `private` *ayuda a los programadores a probar, depurar y modificar correctamente los sistemas. Si una subclase puede acceder a las variables de instancia* `private` *de su superclase, las clases que hereden de esa subclase podrían acceder a las variables de instancia también. Esto propagaría el acceso a las que deberían ser variables de instancia* `private`*, y se perderían los beneficios del ocultamiento de la información.*

9.4 Relación entre las superclases y las subclases

En esta sección usaremos una jerarquía de herencia que contiene tipos de empleados en la aplicación de nómina de una compañía, para hablar sobre la relación entre una superclase y su subclase. En esta compañía, a los empleados por comisión (que se representarán como objetos de una superclase) se les paga un porcentaje de sus ventas, mientras que los empleados por comisión con salario base (que se representarán como objetos de una subclase) reciben un salario base, más un porcentaje de sus ventas.

Dividiremos nuestra discusión sobre la relación entre los empleados por comisión y los empleados por comisión con salario base en cinco ejemplos. El primero declara la clase `EmpleadoPorComision`, la cual hereda directamente de la clase `Object` y declara como variables de instancia `private` el primer nombre, el apellido paterno, el número de seguro social, la tarifa de comisión y el monto de ventas en bruto (es decir, total).

El segundo ejemplo declara la clase `EmpleadoBaseMasComision`, la cual también hereda directamente de la clase `Object` y declara como variables de instancia `private` el primer nombre, el apellido paterno, el número de seguro social, la tarifa de comisión, el monto de ventas en bruto y el salario base. Para crear esta última clase, escribiremos cada línea de código que ésta requiera; pronto veremos que es mucho más eficiente crear esta clase haciendo que herede de la clase `EmpleadoPorComision`.

El tercer ejemplo declara una clase `EmpleadoBaseMasComision2` separada, la cual extiende a la clase `EmpleadoPorComision` (es decir, un `EmpleadoBasePorComision2` *es un* `EmpleadoPorComision` que también tiene un salario base) y trata de acceder a los miembros `private` de la clase `EmpleadoPorComision`; esto produce errores de compilación, ya que la subclase no puede acceder a las variables de instancia `private` de la superclase.

El cuarto ejemplo muestra que si las variables de instancia de `EmpleadoPorComision` se declaran como `protected`, una clase `EmpleadoBaseMasComision3` que extiende a la clase `EmpleadoPorComision2` puede acceder a los datos de manera directa. Para este fin, declaramos la clase `EmpleadoPorComision2` con variables de instancia

`protected`. Todas las clases `EmpleadoBaseMasComision` contienen una funcionalidad idéntica, pero le mostraremos que la clase `EmpleadoBaseMasComision3` es más fácil de crear y de manipular.

Una vez que hablemos sobre la conveniencia de utilizar variables de instancia `protected`, crearemos el quinto ejemplo, el cual establece las variables de instancia de `EmpleadoPorComision` de vuelta a `private` en la clase `EmpleadoPorComision3`, para hacer cumplir la buena ingeniería de software. Después le mostraremos cómo una clase `EmpleadoBaseMasComision4` separada, que extiende a la clase `EmpleadoPorComision3`, puede utilizar los métodos `public` de `EmpleadoPorComision3` para manipular las variables de instancia `private` de `EmpleadoPorComision3`.

9.4.1 Creación y uso de una clase `EmpleadoPorComision`

Comenzaremos por declarar la clase `EmpleadoPorComision` (figura 9.4). La línea 4 empieza la declaración de la clase, e indica que la clase `EmpleadoPorComision` extiende (`extends`) (es decir, hereda de) la clase `Object` (del paquete `java.lang`). Los programadores de Java utilizan la herencia para crear clases a partir de clases existentes. De hecho, todas las clases en Java (excepto `Object`) extienden a una clase existente. Como la clase `EmpleadoPorComision` extiende la clase `Object`, la clase `EmpleadoPorComision` hereda los métodos de la clase `Object`; la clase `Object` no tiene campos. De hecho, cada clase en Java hereda en forma directa o indirecta los métodos de `Object`. Si una clase no especifica que extiende a otra clase, la nueva clase hereda de `Object` en forma implícita. Por esta razón, es común que los programadores no incluyan "`extends Object`" en su código; en nuestro ejemplo lo haremos sólo por fines demostrativos.



Observación de ingeniería de software 9.3

El compilador de Java establece la superclase de una clase a `Object` *cuando la declaración de la clase no extiende explícitamente una superclase.*

```
1  // Fig. 9.4: EmpleadoPorComision.java
2  // La clase EmpleadoPorComision representa a un empleado por comisión.
3
4  public class EmpleadoPorComision extends Object
5  {
6     private String primerNombre;
7     private String apellidoPaterno;
8     private String numeroSeguroSocial;
9     private double ventasBrutas; // ventas semanales totales
10    private double tarifaComision; // porcentaje de comisión
11
12    // constructor con cinco argumentos
13    public EmpleadoPorComision( String nombre, String apellido, String nss,
14       double ventas, double tarifa )
15    {
16       // la llamada implícita al constructor del objeto ocurre aquí
17       primerNombre = nombre;
18       apellidoPaterno = apellido;
19       numeroSeguroSocial = nss;
20       establecerVentasBrutas( ventas ); // valida y almacena las ventas brutas
21       establecerTarifaComision( tarifa ); // valida y almacena la tarifa de comisión
22    } // fin del constructor de EmpleadoPorComision con cinco argumentos
23
24    // establece el primer nombre
25    public void establecerPrimerNombre( String nombre )
26    {
27       primerNombre = nombre;
```

Figura 9.4 | La clase `EmpleadoPorComision` representa a un empleado que recibe como sueldo un porcentaje de las ventas brutas. (Parte 1 de 3).

```
   } // fin del método establecerPrimerNombre

   // devuelve el primer nombre
   public String obtenerPrimerNombre()
   {
      return primerNombre;
   } // fin del método obtenerPrimerNombre

   // establece el apellido paterno
   public void establecerApellidoPaterno( String apellido )
   {
      apellidoPaterno = apellido;
   } // fin del método establecerApellidoPaterno

   // devuelve el apellido paterno
   public String obtenerApellidoPaterno()
   {
      return apellidoPaterno;
   } // fin del método obtenerApellidoPaterno

   // establece el número de seguro social
   public void establecerNumeroSeguroSocial( String nss )
   {
      numeroSeguroSocial = nss; // debe validar
   } // fin del método establecerNumeroSeguroSocial

   // devuelve el número de seguro social
   public String obtenerNumeroSeguroSocial()
   {
      return numeroSeguroSocial;
   } // fin del método obtenerNumeroSeguroSocial

   // establece el monto de ventas totales del empleado por comisión
   public void establecerVentasBrutas( double ventas )
   {
      ventasBrutas = ( ventas < 0.0 ) ? 0.0 : ventas;
   } // fin del método establecerVentasBrutas

   // devuelve el monto de ventas totales del empleado por comisión
   public double obtenerVentasBrutas()
   {
      return ventasBrutas;
   } // fin del método obtenerVentasBrutas

   // establece la tarifa del empleado por comisión
   public void establecerTarifaComision( double tarifa )
   {
      tarifaComision = ( tarifa > 0.0 && tarifa < 1.0 ) ? tarifa : 0.0;
   } // fin del método establecerTarifaComision

   // devuelve la tarifa del empleado por comisión
   public double obtenerTarifaComision()
   {
      return tarifaComision;
   } // fin del método obtenerTarifaComision

   // calcula el salario del empleado por comisión
   public double ingresos()
```

Figura 9.4 | La clase `EmpleadoPorComision` representa a un empleado que recibe como sueldo un porcentaje de las ventas brutas. (Parte 2 de 3).

```
86       {
87          return tarifaComision * ventasBrutas;
88       } // fin del método ingresos
89
90       // devuelve representación String del objeto EmpleadoPorComision
91       public String toString()"
92       {
93          return String.format( "%s: %s %s\n%s: %s\n%s: %.2f\n%s: %.2f",
94             "empleado por comision", primerNombre, apellidoPaterno,
95             "numero de seguro social", numeroSeguroSocial,
96             "ventas brutas", ventasBrutas,
97             “tarifa de comision”, tarifaComision );
98       } // fin del método toString
99    } // fin de la clase EmpleadoPorComision
```

Figura 9.4 | La clase `EmpleadoPorComision` representa a un empleado que recibe como sueldo un porcentaje de las ventas brutas. (Parte 3 de 3).

Los servicios `public` de la clase `EmpleadoPorComision` incluyen un constructor (líneas 13 a 22), y los métodos `ingresos` (líneas 85 a 88) y `toString` (líneas 91 a 98). Las líneas 25 a 82 declaran métodos *establecer* y *obtener* `public` para manipular las variables de instancia `primerNombre`, `apellidoPaterno`, `numeroSeguroSocial`, `ventasBrutas` y `tarifaComision` de la clase (las cuales se declaran en las líneas 6 a 10). La clase `EmpleadoPorComision` declara cada una de sus variables de instancia como `private`, por lo que los objetos de otras clases no pueden acceder directamente a estas variables. Declarar las variables de instancia como `private` y proporcionar métodos *establecer* y *obtener* para manipular y validar las variables de instancia ayuda a cumplir con la buena ingeniería de software. Por ejemplo, los métodos `establecerVentasBrutas` y `establecerTarifaComision` validan sus argumentos antes de asignar los valores a las variables de instancia `ventasBrutas` y `tarifaComision`, en forma respectiva.

Los constructores no se heredan, por lo que la clase `EmpleadoPorComision` no hereda el constructor de la clase `Object`. Sin embargo, el constructor de la clase `EmpleadoPorComision` llama al constructor de la clase `Object` de manera implícita. De hecho, la primera tarea del constructor de cualquier subclase es llamar al constructor de su superclase directa, ya sea en forma explícita o implícita (si no se especifica una llamada al constructor), para asegurar que las variables de instancia heredadas de la superclase se inicialicen en forma apropiada. En la sección 9.4.3 hablaremos sobre la sintaxis para llamar al constructor de una superclase en forma explícita. Si el código no incluye una llamada explícita al constructor de la superclase, Java genera una llamada implícita al constructor predeterminado o sin argumentos de la superclase. El comentario en la línea 16 de la figura 9.4 indica en dónde se hace la llamada implícita al constructor predeterminado de la superclase `Object` (el programador no necesita escribir el código para esta llamada). El constructor predeterminado (vacío) de la clase `Object` no hace nada. Observe que aun si una clase no tiene constructores, el constructor predeterminado que declara el compilador de manera implícita para la clase llamará al constructor predeterminado o sin argumentos de la superclase.

Una vez que se realiza la llamada implícita al constructor de `Object`, las líneas 17 a 21 del constructor de `EmpleadoPorComision` asignan valores a las variables de instancia de la clase. Observe que no validamos los valores de los argumentos `nombre`, `apellido` y `nss` antes de asignarlos a las variables de instancia correspondientes. Podríamos validar el nombre y el apellido; tal vez asegurarnos de que tengan una longitud razonable. De manera similar, podría validarse un número de seguro social, para asegurar que contenga nueve dígitos, con o sin guiones cortos (por ejemplo, `123-45-6789` o `123456789`).

El método `ingresos` (líneas 85 a 88) calcula los ingresos de un `EmpleadoPorComision`. La línea 87 multiplica la `tarifaComision` por las `ventasBrutas` y devuelve el resultado.

El método `toString` (líneas 91 a 98) es especial: es uno de los métodos que hereda cualquier clase de manera directa o indirecta de la clase `Object`, la cual es la raíz de la jerarquía de clases de Java. La sección 9.7 muestra un resumen de los métodos de la clase `Object`. El método `toString` devuelve un `String` que representa a un objeto. Un programa llama a este método de manera implícita cada vez que un objeto debe convertirse en una representación de cadena, como cuando se imprime un objeto mediante `printf` o el método `format` de `String`, usando el especificador de formato `%s`. El método `toString` de la clase `Object` devuelve un `String` que incluye

el nombre de la clase del objeto. En esencia, es un receptáculo que puede sobrescribirse por una subclase para especificar una representación de cadena apropiada de los datos en un objeto de la subclase. El método `toString` de la clase `EmpleadoPorComision` sobrescribe (redefine) al método `toString` de la clase `Object`. Al invocarse, el método `toString` de `EmpleadoPorComision` usa el método `String` llamado `format` para devolver un `String` que contiene información acerca del `EmpleadoPorComision`. Para sobrescribir a un método de una superclase, una subclase debe declarar un método con la misma firma (nombre del método, número de parámetros, tipos de los parámetros y orden de los tipos de los parámetros) que el método de la superclase; el método `toString` de `Object` no recibe parámetros, por lo que `EmpleadoPorComision` declara a `toString` sin parámetros.



Error común de programación 9.1

Es un error de compilación sobrescribir un método con un modificador de acceso más restringido; un método `public` *de la superclase no puede convertirse en un método* `protected` *o* `private` *en la subclase; un método* `protected` *de la superclase no puede convertirse en un método* `private` *en la subclase. Hacer esto sería quebrantar la relación* es un, *en la que se requiere que todos los objetos de la subclase puedan responder a las llamadas a métodos que se hagan a los métodos* `public` *declarados en la superclase. Si un método* `public` *pudiera sobrescribirse como* `protected` *o* `private`*, los objetos de la subclase no podrían responder a las mismas llamadas a métodos que los objetos de la superclase. Una vez que se declara un método como* `public` *en una superclase, el método sigue siendo* `public` *para todas las subclases directas e indirectas de esa clase.*

La figura 9.5 prueba la clase `EmpleadoPorComision`. Las líneas 9 a 10 crean una instancia de un objeto `EmpleadoPorComision` e invocan a su constructor (líneas 13 a 22 de la figura 9.4) para inicializarlo con `"Sue"` como el primer nombre, `"Jones"` como el apellido, `"222-22-2222"` como el número de seguro social, `10000` como el monto de ventas brutas y `.06` como la tarifa de comisión. Las líneas 15 a 24 utilizan los métodos *obtener* de `EmpleadoPorComision` para obtener los valores de las variables de instancia del objeto e imprimirlas en pantalla. Las líneas 26 y 27 invocan a los métodos `establecerVentasBrutas` y `establecerTarifaComision` del objeto para modificar los valores de las variables de instancia `ventasBrutas` y `tarifaComision`. Las líneas 29 y 30 imprimen en pantalla la representación de cadena del `EmpleadoPorComision` actualizado. Observe que cuando se imprime un objeto en pantalla usando el especificador de formato `%s`, se invoca de manera implícita el método `toString` del objeto para obtener su representación de cadena.

```
1  // Fig. 9.5: PruebaEmpleadoPorComision.java
2  // Prueba de la clase EmpleadoPorComision.
3
4  public class PruebaEmpleadoPorComision
5  {
6     public static void main( String args[] )
7     {
8        // crea instancia de objeto EmpleadoPorComision
9        EmpleadoPorComision empleado = new EmpleadoPorComision(
10          "Sue", "Jones", "222-22-2222", 10000, .06 );
11
12       // obtiene datos del empleado por comisión
13       System.out.println(
14          "Informacion del empleado obtenida por los metodos establecer: \n" );
15       System.out.printf( "%s %s\n", "El primer nombre es",
16          empleado.obtenerPrimerNombre() );
17       System.out.printf( "%s %s\n", "El apellido paterno es",
18          empleado.obtenerApellidoPaterno() );
19       System.out.printf( "%s %s\n", "El numero de seguro social es",
20          empleado.obtenerNumeroSeguroSocial() );
21       System.out.printf( "%s %.2f\n", "Las ventas brutas son",
22          empleado.obtenerVentasBrutas() );
23       System.out.printf( "%s %.2f\n", "La tarifa de comision es",
24          empleado.obtenerTarifaComision() );
```

Figura 9.5 | Programa de prueba de la clase `EmpleadoPorComision`. (Parte 1 de 2).

```

      empleado.establecerVentasBrutas( 500 ); // establece las ventas brutas
      empleado.establecerTarifaComision( .1 ); // establece la tarifa de comisión

      System.out.printf( "\n%s:\n\n%s\n",
         "Informacion actualizada del empleado, obtenida mediante toString", empleado );
   } // fin de main
} // fin de la clase PruebaEmpleadoPorComision
```

```
Informacion del empleado obtenida por los metodos establecer:

El primer nombre es Sue
El apellido paterno es Jones
El numero de seguro social es 222-22-2222
Las ventas brutas son 10000.00
La tarifa de comision es 0.06

Informacion actualizada del empleado, obtenida mediante toString:

empleado por comision: Sue Jones
numero de seguro social: 222-22-2222
ventas brutas: 500.00
tarifa de comision: 0.10
```

Figura 9.5 | Programa de prueba de la clase `EmpleadoPorComision`. (Parte 2 de 2).

9.4.2 Creación de una clase `EmpleadoBaseMasComision` sin usar la herencia

Ahora hablaremos sobre la segunda parte de nuestra introducción a la herencia, mediante la declaración y prueba de la clase (completamente nueva e independiente) `EmpleadoBaseMasComision` (figura 9.6), la cual contiene los siguientes datos: primer nombre, apellido paterno, número de seguro social, monto de ventas brutas, tarifa de comisión y salario base. Los servicios `public` de la clase `EmpleadoBaseMasComision` incluyen un constructor (líneas 15 a 25), y los métodos `ingresos` (líneas 100 a 103) y `toString` (líneas 106 a 114). Las líneas 28 a 97 declaran métodos *establecer* y *obtener* `public` para las variables de instancia `private primerNombre, apellidoPaterno`, `numeroSeguroSocial`, `ventasBrutas`, `tarifaComision` y `salarioBase` para la clase (las cuales se declaran en las líneas 7 a 12). Estas variables y métodos encapsulan todas las características necesarias de un empleado por comisión con sueldo base. Observe la similitud entre esta clase y la clase `EmpleadoPorComision` (figura 9.4); en este ejemplo, no explotaremos todavía esa similitud.

```
1  // Fig. 9.6: EmpleadoBaseMasComision.java
2  // La clase EmpleadoBaseMasComision representa a un empleado que recibe
3  // un salario base, además de la comisión.
4
5  public class EmpleadoBaseMasComision
6  {
7     private String primerNombre;
8     private String apellidoPaterno;
9     private String numeroSeguroSocial;
10    private double ventasBrutas; // ventas totales por semana
11    private double tarifaComision; // porcentaje de comisión
12    private double salarioBase; // salario base por semana
13
14    // constructor con seis argumentos
15    public EmpleadoBaseMasComision( String nombre, String apellido,
16       String nss, double ventas, double tarifa, double salario )
17    {
```

Figura 9.6 | La clase `EmpleadoBaseMasComision` representa a un empleado que recibe un salario base, además de la comisión. (Parte 1 de 3).

```
      // la llamada implícita al constructor de Object ocurre aquí
      primerNombre = nombre;
      apellidoPaterno = apellido;
      numeroSeguroSocial = nss;
      establecerVentasBrutas( ventas ); // valida y almacena las ventas brutas
      establecerTarifaComision( tarifa ); // valida y almacena la tarifa de comisión
      establecerSalarioBase( salario ); // valida y almacena el salario base
   } // fin del constructor de EmpleadoBaseMasComision con seis argumentos

   // establece el primer nombre
   public void establecerPrimerNombre( String nombre )
   {
      primerNombre = nombre;
   } // fin del método establecerPrimerNombre

   // devuelve el primer nombre
   public String obtenerPrimerNombre()
   {
      return primerNombre;
   } // fin del método obtenerPrimerNombre

   // establece el apellido paterno
   public void establecerApellidoPaterno( String apellido )
   {
      apellidoPaterno = apellido;
   } // fin del método establecerApellidoPaterno

   // devuelve el apellido paterno
   public String obtenerApellidoPaterno()
   {
      return apellidoPaterno;
   } // fin del método obtenerApellidoPaterno

   // establece el número de seguro social
   public void establecerNumeroSeguroSocial( String nss )
   {
      numeroSeguroSocial = nss; // debe validar
   } // fin del método establecerNumeroSeguroSocial

   // devuelve el número de seguro social
   public String obtenerNumeroSeguroSocial()
   {
      return numeroSeguroSocial;
   } // fin del método obtenerNumeroSeguroSocial

   // establece el monto de ventas brutas
   public void establecerVentasBrutas( double ventas )
   {
      ventasBrutas = ( ventas < 0.0 ) ? 0.0 : ventas;
   } // fin del método establecerVentasBrutas

   // devuelve el monto de ventas brutas
   public double obtenerVentasBrutas()
   {
      return ventasBrutas;
   } // fin del método obtenerVentasBrutas

   // establece la tarifa de comisión
```

Figura 9.6 | La clase `EmpleadoBaseMasComision` representa a un empleado que recibe un salario base, además de la comisión. (Parte 2 de 3).

```
76     public void establecerTarifaComision( double tarifa )
77     {
78        tarifaComision = ( tarifa > 0.0 && tarifa < 1.0 ) ? tarifa : 0.0;
79     } // fin del método establecerTarifaComision
80
81     // devuelve la tarifa de comisión
82     public double obtenerTarifaComision()
83     {
84        return tarifaComision;
85     } // fin del método obtenerTarifaComision
86
87     // establece el salario base
88     public void establecerSalarioBase( double salario )
89     {
90        salarioBase = ( salario < 0.0 ) ? 0.0 : salario;
91     } // fin del método establecerSalarioBase
92
93     // devuelve el salario base
94     public double obtenerSalarioBase()
95     {
96        return salarioBase;
97     } // fin del método obtenerSalarioBase
98
99     // calcula los ingresos
100    public double ingresos()
101    {
102       return salarioBase + ( tarifaComision * ventasBrutas );
103    } //fin del método ingresos
104
105    // devuelve representación String de EmpleadoBaseMasComision
106    public String toString()
107    {
108       return String.format(
109          "%s: %s %s\n%s: %s\n%s: %.2f\n%s: %.2f\n%s: %.2f",
110          "empleado por comision con sueldo base", primerNombre, apellidoPaterno,
111          "numero de seguro social", numeroSeguroSocial,
112          "ventas brutas", ventasBrutas, "tarifa de comision", tarifaComision,
113          "salario base", salarioBase );
114    } // fin del método toString
115 } // fin de la clase EmpleadoBaseMasComision
```

Figura 9.6 | La clase `EmpleadoBaseMasComision` representa a un empleado que recibe un salario base, además de la comisión. (Parte 3 de 3).

Observe que la clase `EmpleadoBaseMasComision` no especifica "`extends Object`" en la línea 5, por lo que la clase extiende a `Object` en forma implícita. Observe además que, al igual que el constructor de la clase `EmpleadoPorComision` (líneas 13 a 22 de la figura 9.4), el constructor de la clase `EmpleadoBaseMasComision` invoca al constructor predeterminado de la clase `Object` en forma implícita, como se indica en el comentario de la línea 18.

El método `ingresos` de la clase `EmpleadoBaseMasComision` (líneas 100 a 103) calcula los ingresos de un empleado por comisión con salario base. La línea 102 devuelve el resultado de sumar el salario base del empleado al producto de multiplicar la tarifa de comisión por las ventas brutas del empleado.

La clase `EmpleadoBaseMasComision` sobrescribe al método `toString` de `Object` para que devuelva un objeto `String` que contiene la información del `EmpleadoBaseMasComision`. Una vez más, utilizamos el especificador de formato `%.2f` para dar formato a las ventas brutas, la tarifa de comisión y el salario base con dos dígitos de precisión a la derecha del punto decimal (línea 109).

La figura 9.7 prueba la clase EmpleadoBaseMasComision. Las líneas 9 a 11 crean una instancia de un objeto EmpleadoBaseMasComision y pasan los argumentos "Bob", "Lewis", "333-33-3333", 5000, .04 y 300 al constructor como el primer nombre, apellido paterno, número de seguro social, ventas brutas, tarifa de comisión y salario base, respectivamente. Las líneas 16 a 27 utilizan los métodos *obtener* de EmpleadoBaseMasComision para obtener los valores de las variables de instancia del objeto e imprimirlos en pantalla. La línea 29 invoca al método establecerSalarioBase del objeto para modificar el salario base. El método establecerSalarioBase (figura 9.6, líneas 88 a 91) asegura que no se le asigne a la variable salarioBase un valor negativo, ya que el salario base de un empleado no puede ser negativo. Las líneas 31 a 33 de la figura 9.7 invocan en forma implícita al método toString del objeto, para obtener su representación de cadena.

La mayor parte del código para la clase EmpleadoBaseMasComision (figura 9.6) es similar, si no es que idéntico, al código para la clase EmpleadoPorComision (figura 9.4). Por ejemplo, en la clase EmpleadoBaseMasComision, las variables de instancia private primerNombre y apellidoPaterno, y los métodos establecerPrimerNombre, obtenerPrimerNombre, establecerApellidoPaterno y obtenerApellidoPaterno son idénticos a los de la clase EmpleadoPorComision. Las clases EmpleadoPorComision y EmpleadoBasePorComision también contienen las variables de instancia private numeroSeguroSocial, tarifaComision y ventasBrutas, así como métodos *obtener* y *establecer* para manipular estas variables. Además, el constructor de EmpleadoBasePorComision es casi idéntico al de la clase EmpleadoPorComision, sólo que el constructor de EmpleadoBaseMasComision también establece el salarioBase. Las demás adiciones a la clase EmpleadoBaseMasComision son la variable de instancia private salarioBase, y los métodos establecerSalarioBase

```
1  // Fig. 9.7: PruebaEmpleadoBaseMasComision.java
2  // Prueba de la clases EmpleadoBaseMasComision.
3
4  public class PruebaEmpleadoBaseMasComision
5  {
6     public static void main( String args[] )
7     {
8        // crea instancia de objeto EmpleadoBaseMasComision
9        EmpleadoBaseMasComision empleado =
10          new EmpleadoBaseMasComision(
11          "Bob", "Lewis", "333-33-3333", 5000, .04, 300 );
12
13       // obtiene datos del empleado por comisión con sueldo base
14       System.out.println(
15          "Informacion del empleado obtenida por metodos establecer: \n" );
16       System.out.printf( "%s %s\n", "El primer nombre es",
17          empleado.obtenerPrimerNombre() );
18       System.out.printf( "%s %s\n", "El apellido es",
19          empleado.obtenerApellidoPaterno() );
20       System.out.printf( "%s %s\n", "El numero de seguro social es",
21          empleado.obtenerNumeroSeguroSocial() );
22       System.out.printf( "%s %.2f\n", "Las ventas brutas son",
23          empleado.obtenerVentasBrutas() );
24       System.out.printf( "%s %.2f\n", "La tarifa de comision es",
25          empleado.obtenerTarifaComision() );
26       System.out.printf( "%s %.2f\n", "El salario base es",
27          empleado.obtenerSalarioBase() );
28
29       empleado.establecerSalarioBase( 1000 ); // establece el salario base
30
31       System.out.printf( "\n%s:\n\n%s\n",
32          "Informacion actualizada del empleado, obtenida por toString",
33          empleado.toString() );
34    } // fin de main
35 } // fin de la clase PruebaEmpleadoBaseMasComision
```

Figura 9.7 | Programa de prueba de EmpleadoBaseMasComision. (Parte 1 de 2).

```
Informacion del empleado obtenida por metodos establecer:

El primer nombre es Bob
El apellido es Lewis
El numero de seguro social es 333-33-3333
Las ventas brutas son 5000.00
La tarifa de comision es 0.04
El salario base es 300.00

Informacion actualizada del empleado, obtenida por toString:

empleado por comision con sueldo base: Bob Lewis
numero de seguro social: 333-33-3333
ventas brutas: 5000.00
tarifa de comision: 0.04
salario base: 1000.00
```

Figura 9.7 | Programa de prueba de `EmpleadoBaseMasComision`. (Parte 2 de 2).

y `obtenerSalarioBase`. El método `toString` de la clase `EmpleadoBaseMasComision` es casi idéntico al de la clase `EmpleadoPorComision`, excepto que el método `toString` de `EmpleadoBasePorComision` también imprime la variable de instancia `salarioBase` con dos dígitos de precisión a la derecha del punto decimal.

Literalmente hablando, copiamos el código de la clase `EmpleadoPorComision` y lo pegamos en la clase `EmpleadoBaseMasComision`, después modificamos esta clase para incluir un salario base y los métodos que manipulan ese salario base. A menudo, este método de "copiar y pegar" está propenso a errores y consume mucho tiempo. Peor aún, se pueden esparcir muchas copias físicas del mismo código a lo largo de un sistema, con lo que el mantenimiento del código se convierte en una pesadilla. ¿Existe alguna manera de "absorber" las variables de instancia y los métodos de una clase, de manera que formen parte de otras clases sin tener que copiar el código? En los siguientes ejemplos responderemos a esta pregunta, utilizando un método más elegante para crear clases, que enfatiza los beneficios de la herencia.



Observación de ingeniería de software 9.4

Copiar y pegar código de una clase a otra puede esparcir los errores a través de varios archivos de código fuente. Para evitar la duplicación de código (y posiblemente los errores) use la herencia o, en algunos casos, la composición, en vez del método de "copiar y pegar", en situaciones en las que desea que una clase "absorba" las variables de instancia y los métodos de otra clase.



Observación de ingeniería de software 9.5

Con la herencia, las variables de instancia y los métodos comunes de todas las clases en la jerarquía se declaran en una superclase. Cuando se requieren modificaciones para estas características comunes, los desarrolladores de software sólo necesitan realizar las modificaciones en la superclase; así las clases derivadas heredan los cambios. Sin la herencia, habría que modificar todos los archivos de código fuente que contengan una copia del código en cuestión.

9.4.3 Creación de una jerarquía de herencia `EmpleadoPorComision-EmpleadoBaseMasComision`

Ahora declararemos la clase `EmpleadoBaseMasComision2` (figura 9.8), que extiende a la clase `EmpleadoPorComision` (figura 9.4). Un objeto `EmpleadoBaseMasComision2` *es un* `EmpleadoPorComision` (ya que la herencia traspasa las capacidades de la clase `EmpleadoPorComision`), pero la clase `EmpleadoBaseMasComision2` también tiene la variable de instancia `salarioBase` (figura 9.8, línea 6). La palabra clave `extends` en la línea 4 de la declaración de la clase indica la herencia. Como subclase, `EmpleadoBaseMasComision2` hereda las variables de instancia `public` y `protected` y los métodos de la clase `EmpleadoPorComision`. El constructor de la clase `EmpleadoPorComision` no se hereda. Por lo tanto, los servicios `public` de `EmpleadoBaseMasComision2` incluyen su constructor (líneas 9 a 16), los métodos `public` heredados de la clase `EmpleadoPorComision`, el

método `establecerSalarioBase` (líneas 19 a 22), el método `obtenerSalarioBase` (líneas 25 a 28), el método `ingresos` (líneas 31 a 35) y el método `toString` (líneas 38 a 47).

El constructor de cada subclase debe llamar en forma implícita o explícita al constructor de su superclase, para asegurar que las variables de instancia heredadas de la superclase se inicialicen en forma apropiada. El constructor de `EmpleadoBaseMasComision2` con seis argumentos (líneas 9 a 16) llama en forma explícita al constructor de la clase `EmpleadoPorComision` con cinco argumentos, para inicializar la porción correspondiente a la superclase de un objeto `EmpleadoBaseMasComision2` (es decir, las variables `primerNombre`, `apellidoPaterno`, `numero-`

```
1  // Fig. 9.8: EmpleadoBaseMasComision2.java
2  // EmpleadoBaseMasComision2 hereda de la clase EmpleadoPorComision.
3
4  public class EmpleadoBaseMasComision2 extends EmpleadoPorComision
5  {
6     private double salarioBase; // salario base por semana
7
8     // constructor con seis argumentos
9     public EmpleadoBaseMasComision2( String nombre, String apellido,
10       String nss, double ventas, double tarifa, double salario )
11    {
12       // llamada explícita al constructor de la superclase EmpleadoPorComision
13       super( nombre, apellido, nss, ventas, tarifa );
14
15       establecerSalarioBase( salario ); // valida y almacena el salario base
16    } // fin del constructor de EmpleadoBaseMasComision2 con seis argumentos
17
18    // establecer salario base
19    public void establecerSalarioBase( double salario )
20    {
21       salarioBase = ( salario < 0.0 ) ? 0.0 : salario;
22    } // fin del método establecerSalarioBase
23
24    // devuelve el salario base
25    public double obtenerSalarioBase()
26    {
27       return salarioBase;
28    } // fin del método obtenerSalarioBase
29
30    // calcula los ingresos
31    public double ingresos()
32    {
33       // no está permitido: tarifaComision y ventasBrutas son private en la superclase
34       return salarioBase + ( tarifaComision * ventasBrutas );
35    } // fin del método ingresos
36
37    // devuelve representación String de EmpleadoBaseMasComision2
38    public String toString()
39    {
40       // no está permitido: intentos por acceder a los miembros private de la superclase
41       return String.format(
42          "%s: %s %s\n%s: %s\n%s: %.2f\n%s: %.2f\n%s: %.2f",
43          "empleado por comision con sueldo base", primerNombre, apellidoPaterno,
44          "numero de seguro social", numeroSeguroSocial,
45          "ventas brutas", ventasBrutas, "tarifa de comision", tarifaComision,
46          "salario base", salarioBase );
47    } // fin del método toString
48 } // fin de la clase EmpleadoBaseMasComision2
```

Figura 9.8 | Los miembros `private` de una superclase no se pueden utilizar en una subclase. (Parte 1 de 2).

```
EmpleadoBaseMasComision2.java:34: tarifaComision has private access in
EmpleadoPorComision
              return salarioBase + ( tarifaComision * ventasBrutas );
                                      ^
EmpleadoBaseMasComision2.java:34: ventasBrutas has private access in
EmpleadoPorComision
       return salarioBase + ( tarifaComision * ventasBrutas );
                                                ^
EmpleadoBaseMasComision2.java:43: primerNombre has private access in
EmpleadoPorComision
        "empleado por comision con sueldo base", primerNombre, apellidoPaterno,
                                                  ^
EmpleadoBaseMasComision2.java:43: apellidoPaterno has private access in
EmpleadoPorComision
        "empleado por comision con sueldo base", primerNombre, apellidoPaterno,
                                                                ^
EmpleadoBaseMasComision2.java:44: numeroSeguroSocial has private access in
EmpleadoPorComision
        "numero de seguro social", numeroSeguroSocial,
                                        ^
EmpleadoBaseMasComision2.java:45: ventasBrutas has private access in
EmpleadoPorComision
        "ventas brutas", ventasBrutas, "tarifa de comision", tarifaComision,
                               ^
EmpleadoBaseMasComision2.java:45: tarifaComision has private access in
EmpleadoPorComision
               "ventas brutas", ventasBrutas, "tarifa de comision", tarifaComision,
                                                                     ^
7 errors
```

Figura 9.8 | Los miembros `private` de una superclase no se pueden utilizar en una subclase. (Parte 2 de 2).

`SeguroSocial`, `ventasBrutas` y `tarifaComision`). La línea 13 en el constructor de `EmpleadoBaseMasComision2` con seis argumentos invoca al constructor de `EmpleadoPorComision` con cinco argumentos (declarado en las líneas 13 a 22 de la figura 9.4) mediante el uso de la **sintaxis de llamada al constructor de la superclase**: la palabra clave `super`, seguida de un conjunto de paréntesis que contienen los argumentos del constructor de la superclase. Los argumentos `nombre`, `apellido`, `nss`, `ventas` y `tarifa` se utilizan para inicializar a los miembros `primerNombre`, `apellidoPaterno`, `numeroSeguroSocial`, `ventasBrutas` y `tarifaComision` de la superclase, respectivamente. Si el constructor de `EmpleadoBaseMasComision2` no invocara al constructor de `EmpleadoPorComision` de manera explícita, Java trataría de invocar al constructor predeterminado o sin argumentos de la clase `EmpleadoPorComision`; pero como la clase no tiene un constructor así, el compilador generaría un error. La llamada explícita al constructor de la superclase en la línea 13 de la figura 9.8 debe ser la primera instrucción en el cuerpo del constructor de la subclase. Además, cuando una superclase contiene un constructor sin argumentos, puede usar a `super()` para llamar a ese constructor en forma explícita, pero esto se hace raras veces.



Error común de programación 9.2

Si el constructor de una subclase llama a uno de los constructores de su superclase con argumentos que no concuerdan exactamente con el número y el tipo de los parámetros especificados en una de las declaraciones del constructor de la clase base, se produce un error de compilación.

El compilador genera errores para la línea 34 de la figura 9.8, debido a que las variables de instancia `tarifaComision` y `ventasBrutas` de la superclase `EmpleadoPorComision` son `private`; no se permite a los métodos de la subclase `EmpleadoBaseMasComision2` acceder a las variables de instancia `private` de la superclase `EmpleadoPorComision`. Observe que utilizamos texto en negritas en la figura 9.8 para indicar que el código es erróneo. El compilador genera errores adicionales en las líneas 43 a 45 del método `toString` de `EmpleadoBaseMasComision2` por la misma razón. Se hubieran podido prevenir los errores en `EmpleadoBaseMasComision2` al utilizar

los métodos *obtener* heredados de la clase `EmpleadoPorComision`. Por ejemplo, la línea 34 podría haber utilizado `obtenerTarifaComision` y `obtenerVentasBrutas` para acceder a las variables de instancia `private tarifaComision` y `ventasBrutas` de `EmpleadoPorComision`, respectivamente. Las líneas 43 a 45 también podrían haber utilizado métodos *establecer* apropiados para obtener los valores de las variables de instancia de la superclase.

9.4.4 La jerarquía de herencia `EmpleadoPorComision`-`EmpleadoBaseMasComision` mediante el uso de variables de instancia `protected`

Para permitir que la clase `EmpleadoBaseMasComision` acceda directamente a las variables de instancia `primerNombre`, `apellidoPaterno`, `numeroSeguroSocial`, `ventasBrutas` y `tarifaComision` de la superclase, podemos declarar esos miembros como `protected` en la superclase. Como vimos en la sección 9.3, los miembros `protected` de una superclase *se* heredan por todas las subclases de esa superclase. La clase `EmpleadoPorComision2` (figura 9.9) es una modificación de la clase `EmpleadoPorComision` (figura 9.4), la cual declara las variables de instancia `primerNombre`, `apellidoPaterno`, `numeroSeguroSocial`, `ventasBrutas` y `tarifaComision` como `protected`, en vez de `private` (figura 9.9, líneas 6 a 10). Aparte del cambio en el nombre de la clase (y por ende el cambio en el nombre del constructor) a `EmpleadoPorComision2`, el resto de la declaración de la clase en la figura 9.9 es idéntico al de la figura 9.4.

```
1  // Fig. 9.9: EmpleadoPorComision2.java
2  // La clase EmpleadoPorComision2 representa a un empleado por comisión.
3
4  public class EmpleadoPorComision2
5  {
6     protected String primerNombre;
7     protected String apellidoPaterno;
8     protected String numeroSeguroSocial;
9     protected double ventasBrutas; // ventas totales por semana
10    protected double tarifaComision; // porcentaje de comisión
11
12    // constructor con cinco argumentos
13    public EmpleadoPorComision2( String nombre, String apellido, String nss,
14       double ventas, double tarifa )
15    {
16       // la llamada implícita al constructor del objeto ocurre aquí
17       primerNombre = nombre;
18       apellidoPaterno = apellido;
19       numeroSeguroSocial = nss;
20       establecerVentasBrutas( ventas ); // valida y almacena las ventas brutas
21       establecerTarifaComision( tarifa ); // valida y almacena la tarifa de comisión
22    } // fin del constructor de EmpleadoPorComision2 con cinco argumentos
23
24    // establece el primer nombre
25    public void establecerPrimerNombre( String nombre )
26    {
27       primerNombre = nombre;
28    } // fin del método establecerPrimerNombre
29
30    // devuelve el primer nombre
31    public String obtenerPrimerNombre()
32    {
33       return primerNombre;
34    } // fin del método obtenerPrimerNombre
35
36    // establece el apellido paterno
```

Figura 9.9 | `EmpleadoPorComision2` con variables de instancia `protected`. (Parte 1 de 3).

```
    public void establecerApellidoPaterno( String apellido )
    {
       apellidoPaterno = apellido;
    } // fin del método establecerApellidoPaterno

    // devuelve el apellido paterno
    public String obtenerApellidoPaterno()
    {
       return apellidoPaterno;
    } // fin del método obtenerApellidoPaterno

    // establece el número de seguro social
    public void establecerNumeroSeguroSocial( String nss )
    {
       numeroSeguroSocial = nss; // debe validar
    } // fin del método establecerNumeroSeguroSocial

    // devuelve el número de seguro social
    public String obtenerNumeroSeguroSocial()
    {
       return numeroSeguroSocial;
    } // fin del método obtenerNumeroSeguroSocial

    // establece el monto de ventas brutas
    public void establecerVentasBrutas( double ventas )
    {
       ventasBrutas = ( ventas < 0.0 ) ? 0.0 : ventas;
    } // fin del método establecerVentasBrutas

    // devuelve el monto de ventas brutas
    public double obtenerVentasBrutas()
    {
       return ventasBrutas;
    } // fin del método obtenerVentasBrutas

    // establece la tarifa de comisión
    public void establecerTarifaComision( double tarifa )
    {
       tarifaComision = ( tarifa > 0.0 && tarifa < 1.0 ) ? tarifa : 0.0;
    } // fin del método establecerTarifaComision

    // devuelve la tarifa de comisión
    public double obtenerTarifaComision()
    {
       return tarifaComision;
    } // fin del método obtenerTarifaComision

    // calcula los ingresos
    public double ingresos()
    {
       return tarifaComision * ventasBrutas;
    } // fin del método ingresos

    // devuelve representación String del objeto EmpleadoPorComision2
    public String toString()
    {
       return String.format( "%s: %s %s\n%s: %s\n%s: %.2f\n%s: %.2f",
          "empleado por comision", primerNombre, apellidoPaterno,
          "numero de seguro social", numeroSeguroSocial,
```

Figura 9.9 | `EmpleadoPorComision2` con variables de instancia `protected`. (Parte 2 de 3).

```
         "ventas brutas", ventasBrutas,
         "tarifa de comision", tarifaComision );
   } // fin del método toString
} // fin de la clase EmpleadoPorComision2
```

Figura 9.9 | `EmpleadoPorComision2` con variables de instancia `protected`. (Parte 3 de 3).

Podríamos haber declarado las variables de instancia `primerNombre`, `apellidoPaterno`, `numeroSeguroSocial`, `ventasBrutas` y `tarifaComision` de la superclase `EmpleadoPorComision2` como `public`, para permitir que la subclase `EmpleadoBaseMasComision2` pueda acceder a las variables de instancia de la superclase. No obstante, declarar variables de instancia `public` es una mala ingeniería de software, ya que permite el acceso sin restricciones a las variables de instancia, lo cual incrementa considerablemente la probabilidad de errores. Con las variables de instancia `protected`, la subclase obtiene acceso a las variables de instancia, pero las clases que no son subclases y las clases que no están en el mismo paquete no pueden acceder a estas variables en forma directa; recuerde que los miembros de clase `protected` son también visibles para las otras clases en el mismo paquete.

La clase `EmpleadoBaseMasComision3` (figura 9.10) es una modificación de la clase `EmpleadoBaseMasComision2` (figura 9.8), que extiende a `EmpleadoPorComision2` (línea 5) en vez de la clase `EmpleadoPorComision`. Los objetos de la clase `EmpleadoBaseMasComision3` heredan las variables de instancia `protected` `primerNombre`, `apellidoPaterno`, `numeroSeguroSocial`, `ventasBrutas` y `tarifaComision` de `EmpleadoPorComision2`; ahora todas estas variables son miembros `protected` de `EmpleadoBaseMasComision3`. Como resultado, el compilador no genera errores al compilar la línea 32 del método `ingresos` y las líneas 40 a 42 del método `toString`. Si otra clase extiende a `EmpleadoBasePorComision3`, la nueva subclase también hereda los miembros `protected`.

```
1  // Fig. 9.10: EmpleadoBaseMasComision3.java
2  // EmpleadoBaseMasComision3 hereda de EmpleadoPorComision2 y tiene
3  // acceso a los miembros protected de EmpleadoPorComision2.
4
5  public class EmpleadoBaseMasComision3 extends EmpleadoPorComision2
6  {
7     private double salarioBase; // salario base por semana
8
9     // constructor con seis argumentos
10    public EmpleadoBaseMasComision3( String nombre, String apellido,
11       String nss, double ventas, double tarifa, double salario )
12    {
13       super( nombre, apellido, nss, ventas, tarifa );
14       establecerSalarioBase( salario ); // valida y almacena el salario base
15    } // fin del constructor de EmpleadoBaseMasComision3 con seis argumentos
16
17    // establece el salario base
18    public void establecerSalarioBase( double salario )
19    {
20       salarioBase = ( salario < 0.0 ) ? 0.0 : salario;
21    } // fin del método establecerSalarioBase
22
23    // devuelve el salario base
24    public double obtenerSalarioBase()
25    {
26       return salarioBase;
27    } // fin del método obtenerSalarioBase
28
```

Figura 9.10 | `EmpleadoBaseMasComision3` hereda las variables de instancia protected de `EmpleadoPorComision3`. (Parte 1 de 2).

```
29     // calcula los ingresos
30     public double ingresos()
31     {
32        return salarioBase + ( tarifaComision * ventasBrutas );
33     } // fin del método ingresos
34
35     // devuelve representación String de EmpleadoBaseMasComision3
36     public String toString()
37     {
38        return String.format(
39           "%s: %s %s\n%s: %s\n%s: %.2f\n%s: %.2f\n%s: %.2f",
40           "empleado por comision con salario base", primerNombre, apellidoPaterno,
41           "numero de seguro social", numeroSeguroSocial,
42           "ventas brutas", ventasBrutas, "tarifa comision", tarifaComision,
43           "salario base", salarioBase );
44     } // fin del método toString
45  } // fin de la clase EmpleadoBaseMasComision3
```

Figura 9.10 | `EmpleadoBaseMasComision3` hereda las variables de instancia protected de `EmpleadoPorComision3`. (Parte 2 de 2).

La clase `EmpleadoBaseMasComision3` no hereda el constructor de la clase `EmpleadoPorComision2`. No obstante, el constructor de la clase `EmpleadoBaseMasComision3` con seis argumentos (líneas 10 a 15) llama al constructor de la clase `EmpleadoPorComision2` con cinco argumentos en forma explícita. El constructor de `EmpleadoBaseMasComision3` con seis argumentos debe llamar en forma explícita al constructor de la clase `EmpleadoPorComision2` con cinco argumentos, ya que `EmpleadoPorComision2` no proporciona un constructor sin argumentos que pueda invocarse en forma implícita.

La figura 9.11 utiliza un objeto `EmpleadoBaseMasComision3` para realizar las mismas tareas que realizó la figura 9.7 con un objeto `EmpleadoBaseMasComision` (figura 9.6). Observe que los resultados de los dos programas son idénticos. Aunque declaramos la clase `EmpleadoBaseMasComision` sin utilizar la herencia, y declaramos la clase `EmpleadoBaseMasComision3` utilizando la herencia, ambas clases proporcionan la misma funcionalidad. El código fuente para la clase `EmpleadoBaseMasComision3`, que tiene 45 líneas, es mucho más corto que el de la clase `EmpleadoBaseMasComision`, que tiene 115 líneas, debido a que la clase `EmpleadoBaseMasComision3` hereda la mayor parte de su funcionalidad de `EmpleadoPorComision2`, mientas que la clase `EmpleadoBaseMasComision` sólo hereda la funcionalidad de la clase `Object`. Además, ahora sólo hay una copia de la funcionalidad del empleado por comisión declarada en la clase `EmpleadoPorComision2`. Esto hace que el código sea más fácil de mantener, modificar y depurar, ya que el código relacionado con un empleado por comisión sólo existe en la clase `EmpleadoPorComision2`.

```
// Fig. 9.11: PruebaEmpleadoBaseMasComision3.java
// Prueba de la clase EmpleadoBaseMasComision3.

public class PruebaEmpleadoBaseMasComision3
{
   public static void main( String args[] )
   {
      // crea instancia de un objeto EmpleadoBaseMasComision3
      EmpleadoBaseMasComision3 empleado =
         new EmpleadoBaseMasComision3(
         "Bob", "Lewis", "333-33-3333", 5000, .04, 300 );

      // obtiene datos del empleado por comision con sueldo base
```

Figura 9.11 | Miembros `protected` de la superclase, heredados en la subclase `EmpleadoBaseMasComision3`. (Parte 1 de 2).

```
      System.out.println(
         "Informacion del empleado, obtenida por los metodos establecer: \n" );
      System.out.printf( "%s %s\n", "El primer nombre es",
         empleadoBaseMasComision.obtenerPrimerNombre() );
      System.out.printf( "%s %s\n", "El apellido es",
         empleadoBaseMasComision.obtenerApellidoPaterno() );
      System.out.printf( "%s %s\n", "El numero de seguro social es",
         empleadoBaseMasComision.obtenerNumeroSeguroSocial() );
      System.out.printf( "%s %.2f\n", "Las ventas brutas son",
         empleadoBaseMasComision.obtenerVentasBrutas() );
      System.out.printf( "%s %.2f\n", "La tarifa de comision es",
         empleadoBaseMasComision.obtenerTarifaComision() );
      System.out.printf( "%s %.2f\n", "El salario base es",
         empleadoBaseMasComision.obtenerSalarioBase() );

      empleadoBaseMasComision.establecerSalarioBase( 1000 ); // establece el salario base

      System.out.printf( "\n%s:\n\n%s\n",
         "Informacion actualizada del empleado, obtenida por toString",
         empleadoBaseMasComision.toString() );
   } // fin de main
} // fin de la clase PruebaEmpleadoBaseMasComision3
```

```
Informacion del empleado, obtenida por los metodos establecer:

El primer nombre es Bob
El apellido es Lewis
El numero de seguro social es 333-33-3333
Las ventas brutas son 5000.00
La tarifa de comision es 0.04
El salario base es 300.00

Informacion actualizada del empleado, obtenida por toString:

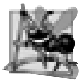
empleado por comision con salario base: Bob Lewis
numero de seguro social: 333-33-3333
ventas brutas: 5000.00
tarifa comision: 0.04
salario base: 1000.00
```

Figura 9.11 | Miembros `protected` de la superclase, heredados en la subclase `EmpleadoBaseMasComision3`. (Parte 2 de 2).

En este ejemplo declaramos las variables de instancia de la superclase como `protected`, para que las subclases pudieran heredarlas. Al heredar variables de instancia `protected` se incrementa un poco el rendimiento, ya que podemos acceder directamente a las variables en la subclase, sin incurrir en la sobrecarga de una llamada a un método *establecer* u *obtener*. No obstante, en la mayoría de los casos es mejor utilizar variables de instancia `private`, para cumplir con la ingeniería de software apropiada, y dejar al compilador las cuestiones relacionadas con la optimización de código. Su código será más fácil de mantener, modificar y depurar.

El uso de variables de instancia `protected` crea varios problemas potenciales. En primer lugar, el objeto de la subclase puede establecer el valor de una variable heredada directamente, sin utilizar un método *establecer*. Por lo tanto, un objeto de la subclase puede asignar un valor inválido a la variable, con lo cual el objeto queda en un estado inconsistente. Por ejemplo, si declaramos la variable de instancia `ventasBrutas` de `EmpleadoPorComision3` como `protected`, un objeto de una subclase (por ejemplo, `EmpleadoBaseMasComision`) podría entonces asignar un valor negativo a `ventasBrutas`. El segundo problema con el uso de variables de instancia `protected` es que hay más probabilidad de que los métodos de la subclase se escriban de manera que dependan de la implementación de datos de la superclase. En la práctica, las subclases sólo deben depender de los servicios de la superclase

(es decir, métodos que no sean `private`) y no en la implementación de datos de la superclase. Si hay variables de instancia `protected` en la superclase, tal vez necesitemos modificar todas las subclases de esa superclase si cambia la implementación de ésta. Por ejemplo, si por alguna razón tuviéramos que cambiar los nombres de las variables de instancia `primerNombre` y `apellidoPaterno` por `nombre` y `apellido`, entonces tendríamos que hacerlo para todas las ocurrencias en las que una subclase haga referencia directa a las variables de instancia `primerNombre` y `apellidoPaterno` de la superclase. En tal caso, se dice que el software es **frágil** o **quebradizo**, ya que un pequeño cambio en la superclase puede "quebrar" la implementación de la subclase. Es conveniente que el programador pueda modificar la implementación de la superclase sin dejar de proporcionar los mismos servicios a las subclases. (Desde luego que, si cambian los servicios de la superclase, debemos reimplementar nuestras subclases). Un tercer problema es que los miembros `protected` de una clase son visibles para todas las clases que se encuentren en el mismo paquete que la clase que contiene los miembros `protected`; esto no siempre es conveniente.



Observación de ingeniería de software 9.6

Use el modificador de acceso `protected` *cuando una superclase deba proporcionar un método sólo a sus subclases y a otras clases en el mismo paquete, pero no a otros clientes.*



Observación de ingeniería de software 9.7

Al declarar variables de instancia `private` *(a diferencia de* `protected`*) en la superclase, se permite que la implementación de la superclase para estas variables de instancia cambie sin afectar las implementaciones de las subclases.*



Tip para prevenir errores 9.1

Siempre que sea posible, no incluya variables de instancia `protected` *en una superclase. En vez de ello, incluya métodos no* `private` *que accedan a las variables de instancia* `private`*. Esto asegurará que los objetos de la clase mantengan estados consistentes.*

9.4.5 La jerarquía de herencia `EmpleadoPorComision`-`EmpleadoBaseMasComision` mediante el uso de variables de `instancia private`

Ahora reexaminaremos nuestra jerarquía una vez más, pero ahora utilizaremos las mejores prácticas de ingeniería de software. La clase `EmpleadoPorComision3` (figura 9.12) declara las variables de instancia `primerNombre`, `apellidoPaterno`, `numeroSeguroSocial`, `ventasBrutas` y `tarifaComision` como `private` (líneas 6 a 10) y proporciona los métodos `public establecerPrimerNombre`, `obtenerPrimerNombre`, `establecerApellidoPaterno`, `obtenerApellidoPaterno`, `establecerNumeroSeguroSocial`, `obtenerNumeroSeguroSocial`, `establecerVentasBrutas`, `obtenerVentasBrutas`, `establecerTarifaComision`, `obtenerTarifaComision`, `ingresos` y `toString` para manipular estos valores. Observe que los métodos `ingresos` (líneas 85 a 88) y `toString` (líneas 91 a 98) utilizan los métodos *obtener* de la clase para obtener los valores de sus variables de instancia. Si decidimos modificar los nombres de las variables de instancia, no habrá que modificar las declaraciones de `ingresos` y de `toString`; sólo habrá que modificar los cuerpos de los métodos *obtener* y *establecer* que manipulan directamente estas variables de instancia. Observe que estos cambios ocurren sólo dentro de la superclase; no se necesitan cambios en la subclase. La localización de los efectos de los cambios como éste es una buena práctica de ingeniería de software. La subclase `EmpleadoBaseMasComision4` (figura 9.13) hereda los miembros no `private` de `EmpleadoPorComision3` y puede acceder a los miembros `private` de su superclase, a través de esos métodos.

La clase `EmpleadoBaseMasComision4` (figura 9.13) tiene varios cambios en las implementaciones de sus métodos, que la diferencian de la clase `EmpleadoBaseMasComision3` (figura 9.10). Los métodos `ingresos` (figura 9.13, líneas 31 a 34) y `toString` (líneas 37 a 41) invocan cada uno al método `obtenerSalarioBase` para obtener el valor del salario base, en vez de acceder en forma directa a `salarioBase`. Si decidimos cambiar el nombre de la variable de instancia `salarioBase`, sólo habrá que modificar los cuerpos de los métodos `establecerSalarioBase` y `obtenerSalarioBase`.

El método `ingresos` de la clase `EmpleadoBaseMasComision4` (figura 9.13, líneas 31 a 34) redefine al método `ingresos` de la clase `EmpleadoPorComision3` (figura 9.12, líneas 85 a 88)para calcular los ingresos de un

```
// Fig. 9.12: EmpleadoPorComision3.java
// La clase EmpleadoPorComision3 representa a un empleado por comisión.

public class EmpleadoPorComision3
{
   private String primerNombre;
   private String apellidoPaterno;
   private String numeroSeguroSocial;
   private double ventasBrutas; // ventas totales por semana
   private double tarifaComision; // porcentaje de comisión

   // constructor con cinco argumentos
   public EmpleadoPorComision3( String nombre, String apellido, String nss,
      double ventas, double tarifa )
   {
      // la llamada implícita al constructor de Object ocurre aquí
      primerNombre = nombre;
      apellidoPaterno = apellido;
      numeroSeguroSocial = nss;
      establecerVentasBrutas( ventas ); // valida y almacena las ventas brutas
      establecerTarifaComision( tarifa ); // valida y almacena la tarifa de comisión
   } // fin del constructor de EmpleadoPorComision3 con cinco argumentos

   // establece el primer nombre
   public void establecerPrimerNombre( String nombre )
   {
      primerNombre = nombre;
   } // fin del método establecerPrimerNombre

   // devuelve el primer nombre
   public String obtenerPrimerNombre()
   {
      return primerNombre;
   } // fin del método obtenerPrimerNombre

   // establece el apellido paterno
   public void establecerApellidoPaterno( String apellido )
   {
      apellidoPaterno = apellido;
   } // fin del método establecerApellidoPaterno

   // devuelve el apellido paterno
   public String obtenerApellidoPaterno()
   {
      return apellidoPaterno;
   } // fin del método obtenerApellidoPaterno

   // establece el número de seguro social
   public void establecerNumeroSeguroSocial( String nss )
   {
      numeroSeguroSocial = nss; // debe validar
   } // fin del método establecerNumeroSeguroSocial

   // devuelve el número de seguro social
   public String obtenerNumeroSeguroSocial()
   {
      return numeroSeguroSocial;
   } // fin del método obtenerNumeroSeguroSocial
```

Figura 9.12 | La clase `EmpleadoPorComision3` utiliza métodos para manipular sus variables de instancia `private`. (Parte 1 de 2).

```

   // establece el monto de ventas brutas
   public void establecerVentasBrutas( double ventas )
   {
      ventasBrutas = ( ventas < 0.0 ) ? 0.0 : ventas;
   } // fin del método establecerVentasBrutas

   // devuelve el monto de ventas brutas
   public double obtenerVentasBrutas()
   {
      return ventasBrutas;
   } // fin del método obtenerVentasBrutas

   // establece la tarifa de comisión
   public void establecerTarifaComision( double tarifa )
   {
      tarifaComision = ( tarifa > 0.0 && tarifa < 1.0 ) ? tarifa : 0.0;
   } // fin del método establecerTarifaComision

   // devuelve la tarifa de comisión
   public double obtenerTarifaComision()
   {
      return tarifaComision;
   } // fin del método obtenerTarifaComision

   // calcula los ingresos
   public double ingresos()
   {
      return obtenerTarifaComision() * obtenerVentasBrutas();
   } // fin del método ingresos

   // devuelve representación String del objeto EmpleadoPorComision3
   public String toString()
   {
      return String.format( “%s: %s %s\n%s: %s\n%s: %.2f\n%s: %.2f”,
         “empleado por comision”, obtenerPrimerNombre(), obtenerApellidoPaterno(),
         “numero de seguro social”, obtenerNumeroSeguroSocial(),
         “ventas brutas”, obtenerVentasBrutas(),
         “tarifa de comision”, obtenerTarifaComision() );
   } // fin del método toString
} // fin de la clase EmpleadoPorComision3
```

Figura 9.12 | La clase `EmpleadoPorComision3` utiliza métodos para manipular sus variables de instancia `private`. (Parte 2 de 2).

empleado por comisión con sueldo base. La nueva versión obtiene la porción de los ingresos del empleado, con base en la comisión solamente, mediante una llamada al método `ingresos` de `EmpleadoPorComision3` con la expresión `super.ingresos()` (figura 9.13, línea 33). El método `ingresos` de `EmpleadoBasePorComision4` suma después el salario base a este valor, para calcular los ingresos totales del empleado. Observe la sintaxis utilizada para invocar un método sobrescrito de la superclase desde una subclase: coloque la palabra clave `super` y un separador `punto` (`.`) antes del nombre del método de la superclase. Esta forma de invocar métodos es una buena práctica de ingeniería de software: en la ***Observación de ingeniería de software 8.5*** vimos que si un método realiza todas o algunas de las acciones que necesita otro método, se hace una llamada a ese método en vez de duplicar su código. Al hacer que el método `ingresos` de `EmpleadoBaseMasComision4` invoque al método `ingresos` de `EmpleadoPorComision3` para calcular parte de los ingresos del objeto `EmpleadoBaseMasComision4`, evitamos duplicar el código y se reducen los problemas de mantenimiento del mismo.

```java
1  // Fig. 9.13: EmpleadoBaseMasComision4.java
2  // La clase EmpleadoBaseMasComision4 hereda de EmpleadoPorComision3 y
3  // accede a los datos private de EmpleadoPorComision3 a través de los
4  // métodos public de EmpleadoPorComision3.
5
6  public class EmpleadoBaseMasComision4 extends EmpleadoPorComision3
7  {
8     private double salarioBase; // salario base por semana
9
10    // constructor con seis argumentos
11    public EmpleadoBaseMasComision4( String nombre, String apellido,
12       String nss, double ventas, double tarifa, double salario )
13    {
14       super( nombre, apellido, nss, ventas, tarifa );
15       establecerSalarioBase( salario ); // valida y almacena el salario base
16    } // fin del constructor de EmpleadoBaseMasComision4 con seis argumentos
17
18    // establece el salario base
19    public void establecerSalarioBase( double salario )
20    {
21       salarioBase = ( salario < 0.0 ) ? 0.0 : salario;
22    } // fin del método establecerSalarioBase
23
24    // devuelve el salario base
25    public double obtenerSalarioBase()
26    {
27       return salarioBase;
28    } // fin del método obtenerSalarioBase
29
30    // calcula los ingresos
31    public double ingresos()
32    {
33       return obtenerSalarioBase() + super.ingresos();
34    } // fin del método ingresos
35
36    // devuelve representación String de EmpleadoBaseMasComision4
37    public String toString()
38    {
39       return String.format( "%s %s\n%s: %.2f", "con sueldo base",
40          super.toString(), "sueldo base", obtenerSalarioBase() );
41    } // fin del método toString
42 } // fin de la clase EmpleadoBaseMasComision4
```

Figura 9.13 | La clase `EmpleadoBaseMasComision4` extiende a `EmpleadoPorComision3`, la cual sólo proporciona variables de instancia `private`.



Error común de programación 9.3

Cuando se sobrescribe un método de la superclase en una subclase, por lo general, la versión correspondiente a la subclase llama a la versión de la superclase para que realice una parte del trabajo. Si no se antepone al nombre del método de la superclase la palabra clave `super` *y el separador punto (*`.`*) cuando se hace referencia al método de la superclase, el método de la subclase se llama a sí mismo, creando potencialmente un error conocido como recursividad infinita. La recursividad, si se utiliza en forma correcta, es una poderosa herramienta, como veremos en el capítulo 15, Recursividad.*

De manera similar, el método `toString` de `EmpleadoBaseMasComision4` (figura 9.13, líneas 37 a 41) sobrescribe el método `toString` de la clase `EmpleadoPorComision3` (figura 9.12, líneas 91 a 98) para devolver una representación `String` apropiada para un empleado por comisión con salario base. La nueva versión crea

parte de la representación `String` de un objeto `EmpleadoBaseMasComision4` (es decir, la cadena `"empleado por comision"` y los valores de las variables de instancia `private` de la clase `EmpleadoPorComision3`), mediante una llamada al método `toString` de `EmpleadoPorComision3` con la expresión `super.toString()` (figura 9.13, línea 40). Después, el método `toString` de `EmpleadoBaseMasComision4` imprime en pantalla el resto de la representación `String` de un objeto `EmpleadoBaseMasComision4` (es decir, el valor del salario base de la clase `EmpleadoBaseMasComision4`).

La figura 9.14 realiza las mismas manipulaciones sobre un objeto `EmpleadoBaseMasComision4` que las de las figuras 9.7 y 9.11 sobre objetos de las clases `EmpleadoBaseMasComision` y `EmpleadoBaseMasComision3`, respectivamente. Aunque cada clase de "empleado por comisión con salario base" se comporta en forma idéntica, la clase `EmpleadoBaseMasComision4` es la mejor diseñada. Mediante el uso de la herencia y las llamadas a métodos que ocultan los datos y aseguran la consistencia, hemos construido una clase bien diseñada con eficiencia y efectividad.

En esta sección vio la evolución de un conjunto de ejemplos diseñados cuidadosamente para enseñar las capacidades clave de la buena ingeniería de software mediante el uso de la herencia. Aprendió a usar la palabra clave `extends` para crear una subclase mediante la herencia, a utilizar miembros `protected` de la superclase para permitir que una subclase acceda a las variables de instancia heredadas de la superclase, y cómo sobrescribir los métodos de la superclase para proporcionar versiones más apropiadas para los objetos de la subclase. Además, aprendió a aplicar las técnicas de ingeniería de software del capítulo 8 y de éste, para crear clases que sean fáciles de mantener, modificar y depurar.

```java
// Fig. 9.14: PruebaEmpleadoBaseMasComision4.java
// Prueba de la clase EmpleadoBaseMasComision4.

public class PruebaEmpleadoBaseMasComision4
{
   public static void main( String args[] )
   {
      // crea instancia de un objeto EmpleadoBaseMasComision4
      EmpleadoBaseMasComision4 empleado =
         new EmpleadoBaseMasComision4(
         "Bob", "Lewis", "333-33-3333", 5000, .04, 300 );

      // obtiene datos del empleado por comisión con sueldo base
      System.out.println(
         "Informacion del empleado obtenida por los metodos establecer: \n" );
      System.out.printf( "%s %s\n", "El primer nombre es",
         empleado.obtenerPrimerNombre() );
      System.out.printf( "%s %s\n", "El apellido es",
         empleado.obtenerApellidoPaterno() );
      System.out.printf( "%s %s\n", "El numero de seguro social es",
         empleado.obtenerNumeroSeguroSocial() );
      System.out.printf( "%s %.2f\n", "Las ventas brutas son",
         empleado.obtenerVentasBrutas() );
      System.out.printf( "%s %.2f\n", "La tarifa de comision es",
         empleado.obtenerTarifaComision() );
      System.out.printf( "%s %.2f\n", "El salario base es",
         empleado.obtenerSalarioBase() );

      empleado.establecerSalarioBase( 1000 ); // establece el salario base

      System.out.printf( "\n%s:\n\n%s\n",
         "Informacion actualizada del empleado, obtenida por toString",
         empleado.toString() );
```

Figura 9.14 | Las variables de instancia `private` de la superclase son accesibles para una subclase, a través de los métodos `public` o `protected` que hereda la subclase. (Parte 1 de 2).

```
    } // fin de main
} // fin de la clase PruebaEmpleadoBaseMasComision4
```

```
Informacion del empleado obtenida por los metodos establecer:

El primer nombre es Bob
El apellido es Lewis
El numero de seguro social es 333-33-3333
Las ventas brutas son 5000.00
La tarifa de comision es 0.04
El salario base es 300.00

Informacion actualizada del empleado, obtenida por toString:

con sueldo base empleado por comision: Bob Lewis
numero de seguro social: 333-33-3333
ventas brutas: 5000.00
tarifa de comision: 0.04
sueldo base: 1000.00
```

Figura 9.14 | Las variables de instancia `private` de la superclase son accesibles para una subclase, a través de los métodos `public` o `protected` que hereda la subclase. (Parte 2 de 2).

9.5 Los constructores en las subclases

Como explicamos en la sección anterior, al crear una instancia de un objeto de una subclase se empieza una cadena de llamadas a los constructores, en los que el constructor de la subclase, antes de realizar sus propias tareas, invoca al constructor de su superclase, ya sea en forma explícita (por medio de la referencia `super`) o implícita (llamando al constructor predeterminado o sin argumentos de la superclase). De manera similar, si la superclase se deriva de otra clase (como sucede con cualquier clase, excepto `Object`), el constructor de la superclase invoca al constructor de la siguiente clase que se encuentre a un nivel más arriba en la jerarquía, y así en lo sucesivo. El último constructor que se llama en la cadena es siempre el de la clase `Object`. El cuerpo del constructor de la subclase original termina de ejecutarse al último. El constructor de cada superclase manipula las variables de instancia de la superclase que hereda el objeto de la subclase. Por ejemplo, considere de nuevo la jerarquía `EmpleadoPorComision3`–`EmpleadoBaseMasComision4` de las figuras 9.12 y 9.13. Cuando un programa crea un objeto `EmpleadoBaseMasComision4`, se hace una llamada al constructor de `EmpleadoBaseMasComision4`. Ese constructor llama al constructor de la clase `EmpleadoPorComision3`, que a su vez llama en forma implícita al constructor de `Object`. El constructor de la clase `Object` tiene un cuerpo vacío, por lo que devuelve de inmediato el control al constructor de `EmpleadoPorComision3`, el cual inicializa las variables de instancia `private` de `EmpleadoPorComision3` que son parte del objeto `EmpleadoBaseMasComision4`. Cuando este constructor termina de ejecutarse, devuelve el control al constructor de `EmpleadoBaseMasComision4`, el cual inicializa el `salarioBase` del objeto `EmpleadoBaseMasComision4`.



Observación de ingeniería de software 9.8

Cuando un programa crea un objeto de una subclase, el constructor de la subclase llama de inmediato al constructor de la superclase (ya sea en forma explícita, mediante `super`*, o implícita). El cuerpo del constructor de la superclase se ejecuta para inicializar las variables de instancia de la superclase que forman parte del objeto de la subclase, después se ejecuta el cuerpo del constructor de la subclase para inicializar las variables de instancia que son parte sólo de la subclase. Java asegura que, aún si un constructor no asigna un valor a una variable de instancia, la variable de todas formas se inicializa con su valor predeterminado (es decir,* `0` *para los tipos numéricos primitivos,* `false` *para los tipos* `boolean` *y* `null` *para las referencias).*

En nuestro siguiente ejemplo volvemos a utilizar la jerarquía de empleado por comisión, al declarar las clases `EmpleadoPorComision4` (figura 9.15) y `EmpladoBaseMasComision5` (figura 9.16). El constructor de cada clase imprime un mensaje cuando se le invoca, lo cual nos permite observar el orden en el que se ejecutan los constructores en la jerarquía.

```
// Fig. 9.15: EmpleadoPorComision4.java
// La clase EmpleadoPorComision4 representa a un empleado por comisión.

public class EmpleadoPorComision4
{
   private String primerNombre;
   private String apellidoPaterno;
   private String numeroSeguroSocial;
   private double ventasBrutas; // ventas totales por semana
   private double tarifaComision; // porcentaje de comisión

   // constructor con cinco argumentos
   public EmpleadoPorComision4( String nombre, String apellido, String nss,
      double ventas, double tarifa )
   {
      // la llamada implícita al constructor de Object ocurre aquí
      primerNombre = nombre;
      apellidoPaterno = apellido;
      numeroSeguroSocial = nss;
      establecerVentasBrutas( ventas ); // valida y almacena las ventas brutas
      establecerTarifaComision( tarifa ); // valida y almacena la tarifa de comisión

      System.out.printf(
         "\nConstructor de EmpleadoPorComision4:\n%s\n", this );
   } // fin del constructor de EmpleadoPorComision4 con cinco argumentos

   // establece el primer nombre
   public void establecerPrimerNombre( String nombre )
   {
      primerNombre = nombre;
   } // fin del método establecerPrimerNombre

   // devuelve el primer nombre
   public String obtenerPrimerNombre()
   {
      return primerNombre;
   } // fin del método obtenerPrimerNombre

   // establece el apellido paterno
   public void establecerApellidoPaterno( String apellido )
   {
      apellidoPaterno = apellido;
   } // fin del método establecerApellidoPaterno

   // devuelve el apellido paterno
   public String obtenerApellidoPaterno()
   {
      return apellidoPaterno;
   } // fin del método obtenerApellidoPaterno

   // establece el número de seguro social
   public void establecerNumeroSeguroSocial( String nss )
   {
      numeroSeguroSocial = nss; // debe validar
   } // fin del método establecerNumeroSeguroSocial

   // devuelve el número de seguro social
   public String obtenerNumeroSeguroSocial()
   {
```

Figura 9.15 | El constructor de `EmpleadoPorComision4` imprime texto en pantalla. (Parte 1 de 2).

```
60          return numeroSeguroSocial;
61       } // fin del método obtenerNumeroSeguroSocial
62
63       // establece el monto de ventas brutas
64       public void establecerVentasBrutas( double ventas )
65       {
66          ventasBrutas = ( ventas < 0.0 ) ? 0.0 : ventas;
67       } // fin del método establecerVentasBrutas
68
69       // devuelve el monto de ventas brutas
70       public double obtenerVentasBrutas()
71       {
72          return ventasBrutas;
73       } // fin del método obtenerVentasBrutas
74
75       // establece la tarifa de comisión
76       public void establecerTarifaComision( double tarifa )
77       {
78          tarifaComision = ( tarifa > 0.0 && tarifa < 1.0 ) ? tarifa : 0.0;
79       } // fin del método establecerTarifaComision
80
81       // devuelve la tarifa de comisión
82       public double obtenerTarifaComision()
83       {
84          return tarifaComision;
85       } // fin del método obtenerTarifaComision
86
87       // calcula los ingresos
88       public double ingresos()
89       {
90          return obtenerTarifaComision() * obtenerVentasBrutas();
91       } // fin del método ingresos
92
93       // devuelve representación String del objeto EmpleadoPorComision4
94       public String toString()
95       {
96          return String.format( "%s: %s %s\n%s: %s\n%s: %.2f\n%s: %.2f",
97             "empleado por comision", obtenerPrimerNombre(), obtenerApellidoPaterno(),
98             "numero de seguro social", obtenerNumeroSeguroSocial(),
99             "ventas brutas", obtenerVentasBrutas(),
100            "tarifa de comision", obtenerTarifaComision() );
101      } // fin del método toString
102   } // fin de la clase EmpleadoPorComision4
```

Figura 9.15 | El constructor de `EmpleadoPorComision4` imprime texto en pantalla. (Parte 2 de 2).

La clase `EmpleadoPorComision4` (figura 9.15) contiene las mismas características que la versión de la clase que se muestra en la figura 9.4. Modificamos el constructor (líneas 13 a 25) para imprimir texto en pantalla al momento en que se invoca. Observe que si imprimimos `this` en pantalla con el especificador de formato `%s` (líneas 23 y 24), invocamos en forma implícita al método `toString` del objeto que se está creando, para obtener la representación `String` de ese objeto.

La clase `EmpleadoBaseMasComision5` (figura 9.16) es casi idéntica a `EmpleadoBaseMasComision4` (figura 9.13), sólo que el constructor de `EmpleadoBaseMasComision5` también imprime texto cuando se invoca. Al igual que en `EmpleadoPorComision4` (figura 9.15), imprimimos el valor de `this` en pantalla usando el especificador de formato `%s` (línea 16), para obtener de manera implícita la representación `String` del objeto.

La figura 9.17 demuestra el orden en el que se llaman los constructores para los objetos de las clases que forman parte de una jerarquía de herencia. El método `main` empieza por crear una instancia del objeto `empleado1` de la clase `EmpleadoPorComision4` (líneas 8 y 9). A continuación, las líneas 12 a 14 crean una instancia del

```java
// Fig. 9.16: EmpleadoBaseMasComision5.java
// Declaración de la clase EmpleadoBaseMasComision5.

public class EmpleadoBaseMasComision5 extends EmpleadoPorComision4
{
   private double salarioBase; // salario base por semana

   // constructor con seis argumentos
   public EmpleadoBaseMasComision5( String nombre, String apellido,
      String nss, double ventas, double tarifa, double salario )
   {
      super( nombre, apellido, nss, ventas, tarifa );
      establecerSalarioBase( salario ); // valida y almacena el salario base

      System.out.printf(
         "\nConstructor de EmpleadoBaseMasComision5:\n%s\n", this );
   } // fin del constructor de EmpleadoBaseMasComision5 con seis argumentos

   // establece el salario base
   public void establecerSalarioBase( double salario )
   {
      salarioBase = ( salario < 0.0 ) ? 0.0 : salario;
   } // fin del método establecerSalarioBase

   // devuelve el salario base
   public double obtenerSalarioBase()
   {
      return salarioBase;
   } // fin del método obtenerSalarioBase

   // calcula los ingresos
   public double ingresos()
   {
      return obtenerSalarioBase() + super.ingresos();
   } // fin del método ingresos

   // devuelve representación String de EmpleadoBaseMasComision5
   public String toString()
   {
      return String.format( "%s %s\n%s: %.2f", "con sueldo base",
         super.toString(), "sueldo base", obtenerSalarioBase() );
   } // fin del método toString
} // fin de la clase EmpleadoBaseMasComision5
```

Figura 9.16 | El constructor de `EmpleadoBaseMasComision5` imprime texto en pantalla.

```java
// Fig. 9.17: PruebaConstructores.java
// Muestra el orden en el que se llaman los constructores de la superclase y la subclase.

public class PruebaConstructores
{
   public static void main( String args[] )
   {
      EmpleadoPorComision4 empleado1 = new EmpleadoPorComision4(
         "Bob", "Lewis", "333-33-3333", 5000, .04 );

      System.out.println();
```

Figura 9.17 | Orden de llamadas a los constructores. (Parte 1 de 2).

```
12         EmpleadoBaseMasComision5 empleado2 =
13            new EmpleadoBaseMasComision5(
14            "Lisa", "Jones", "555-55-5555", 2000, .06, 800 );
15
16         System.out.println();
17         EmpleadoBaseMasComision5 empleado3 =
18            new EmpleadoBaseMasComision5(
19            "Mark", "Sands", "888-88-8888", 8000, .15, 2000 );
20      } // fin de main
21   } // fin de la clase PruebaConstructores
```

```
Constructor de EmpleadoPorComision4:
empleado por comision: Bob Lewis
numero de seguro social: 333-33-3333
ventas brutas: 5000.00
tarifa de comision: 0.04


Constructor de EmpleadoPorComision4:
con sueldo base empleado por comision: Lisa Jones
numero de seguro social: 555-55-5555
ventas brutas: 2000.00
tarifa de comision: 0.06
sueldo base: 0.00

Constructor de EmpleadoBaseMasComision5:
con sueldo base empleado por comision: Lisa Jones
numero de seguro social: 555-55-5555
ventas brutas: 2000.00
tarifa de comision: 0.06
sueldo base: 800.00


Constructor de EmpleadoPorComision4:
con sueldo base empleado por comision: Mark Sands
numero de seguro social: 888-88-8888
ventas brutas: 8000.00
tarifa de comision: 0.15
sueldo base: 0.00

Constructor de EmpleadoBaseMasComision5:
con sueldo base empleado por comision: Mark Sands
numero de seguro social: 888-88-8888
ventas brutas: 8000.00
tarifa de comision: 0.15
sueldo base: 2000.00
```

Figura 9.17 | Orden de llamadas a los constructores. (Parte 2 de 2).

objeto `empleado2` de `EmpleadoBaseMasComision5`. Esto invoca al constructor de `EmpleadoPorComision4`, el cual imprime los resultados con los valores que recibe del constructor de `EmpleadoBaseMasComision5` y después imprime los resultados especificados en el constructor de `EmpleadoBaseMasComision5`. Después, las líneas 17 a 19 crean una instancia del objeto `empleado3` de `EmpleadoBaseMasComision5`. De nuevo, se hacen llamadas a los constructores de `EmpleadoPorComision4` y `EmpleadoBaseMasComision5`. En cada caso, el cuerpo del constructor de `EmpleadoPorComision4` se ejecuta antes que el cuerpo del constructor de `EmpleadoBaseMasComision5`. Observe que `empleado2` se construye por completo antes que empiece el constructor de `empleado3`.

9.6 Ingeniería de software mediante la herencia

En esta sección hablaremos sobre la personalización del software existente mediante la herencia. Cuando una nueva clase extiende a una clase existente, la nueva clase hereda los miembros no `private` de la clase existente. Podemos personalizar la nueva clase para cumplir nuestras necesidades, mediante la inclusión de miembros adicionales y la sobrescritura de miembros de la superclase. Para hacer esto, el programador de la subclase4 no tiene que modificar el código fuente de la superclase. Java sólo requiere el acceso al archivo `.class` de la superclase, para poder compilar y ejecutar cualquier programa que utilice o extienda la superclase. Esta poderosa capacidad es atractiva para los distribuidores independientes de software (ISVs), quienes pueden desarrollar clases propietarias para vender o licenciar, y ponerlas a disposición de los usuarios en formato de código de bytes. Después, los usuarios pueden derivar con rapidez nuevas clases a partir de estas clases de biblioteca, sin necesidad de acceder al código fuente propietario del ISV.



Observación de ingeniería de software 9.9

A pesar del hecho de que al heredar de una clase no se requiere acceso al código fuente de esa clase, los desarrolladores insisten con frecuencia en ver el código fuente para comprender cómo está implementada la clase. Los desarrolladores en la industria desean asegurarse que están extendiendo una clase sólida; por ejemplo, una clase que se desempeñe bien y que se implemente en forma segura.

Algunas veces, los estudiantes tienen dificultad para apreciar el alcance de los problemas a los que se enfrentan los diseñadores que trabajan en proyectos de software a gran escala en la industria. Las personas experimentadas con esos proyectos dicen que la reutilización efectiva del software mejora el proceso de desarrollo del mismo. La programación orientada a objetos facilita la reutilización de software, con lo que se obtiene una potencial reducción en el tiempo de desarrollo.

La disponibilidad de bibliotecas de clases extensas y útiles produce los máximos beneficios de la reutilización de software a través de la herencia. Los diseñadores de aplicaciones crean sus aplicaciones con estas bibliotecas, y los diseñadores de bibliotecas obtienen su recompensa al incluir sus bibliotecas con las aplicaciones. Las bibliotecas de clases estándar de Java que se incluyen con Java SE 6 tienden a ser de propósito general. Existen muchas bibliotecas de clases de propósito especial, y muchas más están en proceso de crearse.



Observación de ingeniería de software 9.10

En la etapa de diseño de un sistema orientado a objetos, el diseñador encuentra comúnmente que ciertas clases están muy relacionadas. Es conveniente que el diseñador "factorice" las variables de instancia y los métodos comunes, y los coloque en una superclase. Después debe usar la herencia para desarrollar subclases, especializándolas con herramientas que estén más allá de las heredadas de parte de la superclase.



Observación de ingeniería de software 9.11

Declarar una subclase no afecta el código fuente de la superclase. La herencia preserva la integridad de la superclase.



Observación de ingeniería de software 9.12

Así como los diseñadores de sistemas no orientados a objetos deben evitar la proliferación de métodos, los diseñadores de sistemas orientados a objetos deben evitar la proliferación de clases. Dicha proliferación crea problemas administrativos y puede obstaculizar la reutilización de software, ya que en una biblioteca de clases enorme es difícil para un cliente localizar las clases más apropiadas. La alternativa es crear menos clases que proporcionen una funcionalidad más substancial, pero dichas clases podrían volverse complejas.



Tip de rendimiento 9.1

Si las subclases son más grandes de lo necesario (es decir, que contengan demasiada funcionalidad), podrían desperdiciarse los recursos de memoria y de procesamiento. Extienda la superclase que contenga la funcionalidad que esté más cerca de lo que usted necesita.

Puede ser confuso leer las declaraciones de las subclases, ya que los miembros heredados no se declaran de manera explícita en las subclases, sin embargo, están presentes en ellas. Hay un problema similar a la hora de documentar los miembros de las subclases.

9.7 La clase object

Como vimos al principio en este capítulo, todas las clases en Java heredan, ya sea en forma directa o indirecta de la clase `Object` (paquete `java.lang`), por lo que todas las demás clases heredan sus 11 métodos. La figura 9.18 muestra un resumen de los métodos de `Object`.

A lo largo de este libro veremos varios de los métodos de `Object` (como se indica en la figura 9.18). Puede aprender más acerca de los métodos de `Object` en la documentación en línea de la API de `Object`, y en el tutorial de Java (*The Java Tutorial*) en los siguientes sitios:

```
java.sun.com/javase/6/docs/api/java/lang/Object.html
java.sun.com/docs/books/tutorial/java/IandI/objectclass.html
```

En el capítulo 7 vimos que los arreglos son objetos. Como resultado, al igual que otros objetos, un arreglo hereda los miembros de la clase `Object`. Observe que todo arreglo tiene un método `clone` sobrescrito, que copia el arreglo. No obstante, si el arreglo almacena referencias a objetos, los objetos no se copian. Para obtener más información acerca de la relación entre los arreglos y la clase `Object`, por favor consulte la *Especificación del lenguaje Java, capítulo 10*, en

```
java.sun.com/docs/books/jls/second_edition/html/arrays.doc.html
```

Método	Descripción
`clone`	Este método `protected`, que no recibe argumentos y devuelve una referencia `Object`, realiza una copia del objeto en el que se llama. Cuando se requiere la clonación para los objetos de una clase, ésta debe sobrescribir el método `clone` como un método `public`, y debe implementar la interfaz `Cloneable` (paquete `java.lang`). La implementación predeterminada de este método realiza algo que se conoce como **copia superficial**: los valores de las variables de instancia en un objeto se copian a otro objeto del mismo tipo. Para los tipos por referencia, sólo se copian las referencias. Una implementación típica del método `clone` sobrescrito sería realizar una **copia en profundidad**, que crea un nuevo objeto para cada variable de instancia de tipo por referencia. Hay muchos detalles sutiles en cuanto a sobrescribir el método `clone`. Puede aprender más acerca de la clonación en el siguiente artículo: `java.sun.com/developer/JDCTechTips/2001/tt0306.html`
`equals`	Este método compara la igualdad entre dos objetos; devuelve `true` si son iguales y `false` en caso contrario. El método recibe cualquier objeto `Object` como argumento. Cuando debe compararse la igualdad entre objetos de una clase en particular, la clase debe sobrescribir el método `equals` para comparar el contenido de los dos objetos. La implementación de este método debe cumplir los siguientes requerimientos: • Debe devolver `false` si el argumento es `null`. • Debe devolver `true` si un objeto se compara consigo mismo, como en `objeto1.equals( objeto1 )`. • Debe devolver `true` sólo si tanto `objeto1.equals( objeto2 )` como `objeto2.equals( objeto1 )` devuelven `true`. • Para tres objetos, si `objeto1.equals( objeto2 )` devuelve `true` y `objeto2.equals( objeto3 )` devuelve `true`, entonces `objeto1.equals( objeto3 )` también debe devolver `true`. • Si `equals` se llama varias veces con los dos objetos, y éstos no cambian, el método debe devolver `true` de manera consistente si los objetos son iguales, y `false` en caso contrario. Una clase que sobrescribe a `equals` también debe sobrescribir `hashCode` para asegurar que los objetos iguales tengan códigos de hash idénticos. La implementación `equals` predeterminada utiliza el operador == para determinar si dos referencias *se refieren al mismo objeto* en la memoria. La sección 30.3.3 demuestra el método `equals` de la clase `String` y explica la diferencia entre comparar objetos `String` con == y con `equals`.

Figura 9.18 | Los métodos de `Object` que todas las clases heredan en forma directa o indirecta. (Parte 1 de 2).

Método	Descripción
`finalize`	El recolector de basura llama a este método `protected` (presentado en las secciones 8.10 y 8.11) para realizar las tareas de preparación para la terminación en un objeto, justo antes de que el recolector de basura reclame la memoria de ese objeto. No se garantiza que el recolector de basura vaya a reclamar un objeto, por lo que no se puede garantizar que se ejecute el método `finalize` del objeto. El método debe especificar una lista de parámetros vacía y debe devolver `void`. La implementación predeterminada de este método sirve como un receptáculo que no hace nada.
`getClass`	Todo objeto en Java conoce su tipo en tiempo de ejecución. El método `getClass` (utilizado en las secciones 10.5 y 21.3) devuelve un objeto de la clase `Class` (paquete `java.lang`), el cual contiene información acerca del tipo del objeto, como el nombre de su clase (devuelto por el método `getName` de `Class`). Puede aprender más acerca de la clase `Class` en la documentación de la API en línea, en `java.sun.com/javase/6/docs/api/java/lang/Class.html`.
`hashCode`	Una tabla de hash es una estructura de datos (descrita en la sección 19.10) que relaciona a un objeto, llamado la clave, con otro objeto, llamado el valor. Cuando inicialmente se inserta un valor en una tabla de hash, se hace una llamada al método `hashCode` de la clave. La tabla de hash utiliza el valor de código de hash devuelto para determinar la ubicación en la que se debe insertar el valor correspondiente. La tabla de hash también utiliza el código de hash de la clave para localizar el valor correspondiente de la misma.
`notify`, `notifyAll`, `wait`	Los métodos `notify`, `notifyAll` y las tres versiones sobrecargadas de `wait` están relacionados con el subprocesamiento múltiple, que veremos en el capítulo 23. En versiones recientes de Java, el modelo de subprocesamiento múltiple ha cambiado en forma considerable, pero estas características se siguen soportando.
`toString`	Este método (presentado en la sección 9.4.1) devuelve una representación `String` de un objeto. La implementación predeterminada de este método devuelve el nombre del paquete y el nombre de la clase del objeto, seguidos por una representación hexadecimal del valor devuelto por el método `hashCode` del objeto.

Figura 9.18 | Los métodos de `Object` que todas las clases heredan en forma directa o indirecta. (Parte 2 de 2).

9.8 (Opcional) Ejemplo práctico de GUI y gráficos: mostar texto e imágenes usando etiquetas

A menudo, los programas usan etiquetas cuando necesitan mostrar información o instrucciones al usuario, en una interfaz gráfica de usuario. Las **etiquetas** son una forma conveniente de identificar componentes de la GUI en la pantalla, y de mantener al usuario informado acerca del estado actual del programa. En Java, un objeto de la clase `JLabel` (del paquete `javax.swing`) puede mostrar una sola línea de texto, una imagen o ambos. El ejemplo de la figura 9.19 demuestra varias características de `JLabel`.

Las líneas 3 a 6 importan las clases que necesitamos para mostrar los objetos `JLabel.BorderLayout` del paquete `java.awt` contienen constantes que especifican en dónde podemos colocar componentes de GUI en el objeto `JFrame`. La clase `ImageIcon` representa una imagen que puede mostrarse en un `JLabel`, y la clase `JFrame` representa la ventana que contiene todas las etiquetas.

La línea 13 crea un objeto `JLabel` que muestra el argumento de su constructor: la cadena `"Norte"`. La línea 16 declara la variable local `etiquetaIcono` y le asigna un nuevo objeto `ImageIcon`. El constructor para `ImageIcon` recibe un objeto `String` que especifica la ruta del archivo de la imagen. Como sólo especificamos un nombre de archivo, Java supone que se encuentra en el mismo directorio que la clase `DemoLabel`. `ImageIcon` puede cargar imágenes en los formatos GIF, JPEG y PNG. La línea 19 declara e inicializa la variable local `etiquetaCentro` con un objeto `JLabel` que muestra el objeto `etiquetaIcono`. La línea 22 declara e inicializa la variable local `etiquetaSur` con un objeto `JLabel` similar al de la línea 19. Sin embargo, la línea 25 llama al método `setText` para modificar el texto que muestra la etiqueta. El método `setText` puede llamarse en cualquier objeto `JLabel` para modificar su texto. Este objeto `JLabel` muestra tanto el icono como el texto.

```java
// Fig 9.19: DemoLabel.java
// Demuestra el uso de etiquetas.
import java.awt.BorderLayout;
import javax.swing.ImageIcon;
import javax.swing.JLabel;
import javax.swing.JFrame;

public class DemoLabel
{
   public static void main( String args[] )
   {
      // Crea una etiqueta con texto solamente
      JLabel etiquetaNorte = new JLabel( "Norte" );

      // crea un icono a partir de una imagen, para poder colocarla en un objeto JLabel
      ImageIcon etiquetaIcono = new ImageIcon( "GUItip.gif" );

      // crea una etiqueta con un icono en vez de texto
      JLabel etiquetaCentro = new JLabel( etiquetaIcono );

      // crea otra etiqueta con un icono
      JLabel etiquetaSur = new JLabel( etiquetaIcono );

      // establece la etiqueta para mostrar texto (así como un icono)
      etiquetaSur.setText( "Sur" );

      // crea un marco para contener las etiquetas
      JFrame aplicacion = new JFrame();

      aplicacion.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );

      // agrega las etiquetas al marco; el segundo argumento especifica
      // en qué parte del marco se va a agregar la etiqueta
      aplicacion.add( etiquetaNorte, BorderLayout.NORTH );
      aplicacion.add( etiquetaCentro, BorderLayout.CENTER );
      aplicacion.add( etiquetaSur, BorderLayout.SOUTH );

      aplicacion.setSize( 300, 300 ); // establece el tamaño del marco
      aplicacion.setVisible( true ); // muestra el marco
   } // fin de main
} // fin de la clase DemoLabel
```

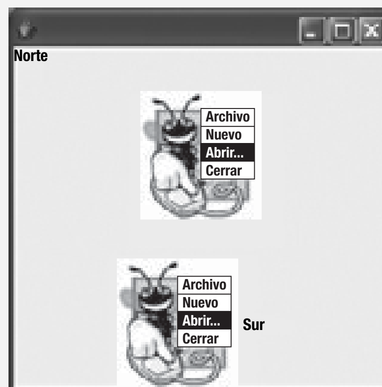


Figura 9.19 | `JLabel` con texto y con imágenes.

La línea 28 crea el objeto `JFrame` que muestra a los objetos `JLabel`, y la línea 30 indica que el programa debe terminar cuando se cierre el objeto `JFrame`. Para adjuntar las etiquetas al objeto `JFrame` en las líneas 34 a 36, llamamos a una versión sobrecargada del método `add` que recibe dos parámetros. El primer parámetro es el componente que deseamos adjuntar, y el segundo es la región en la que debe colocarse. Cada objeto `JFrame` tiene un **esquema** asociado, que ayuda al `JFrame` a posicionar los componentes de la GUI que tiene adjuntos. El esquema predeterminado para un objeto `JFrame` se conoce como `BorderLayout`, y tiene cinco regiones: `NORTH` (superior), `SOUTH` (inferior), `EAST` (lado derecho), `WEST` (lado izquierdo) y `CENTER` (centro). Cada una de estas regiones se declara como una constante en la clase `BorderLayout`. Al llamar al método `add` con un argumento, el objeto `JFrame` coloca el componente en la región `CENTER` de manera automática. Si una posición ya contiene un componente, entonces el nuevo componente toma su lugar. Las líneas 38 y 39 establecen el tamaño del objeto `JFrame` y lo hacen visible en pantalla.

Ejercicio del ejemplo práctico de GUI y gráficos

9.1 Modifique el ejercicio 8.1 para incluir un objeto `JLabel` como barra de estado, que muestre las cuentas que representan el número de cada figura mostrada. La clase `PanelDibujo` debe declarar un método que devuelva un objeto `String` que contenga el texto de estado. En `main`, primero cree el objeto `PanelDibujo`, y después cree el objeto `JLabel` con el texto de estado como argumento para el constructor de `JLabel`. Adjunte el objeto `JLabel` a la región SOUTH del objeto `JFrame`, como se muestra en la figura 9.20.

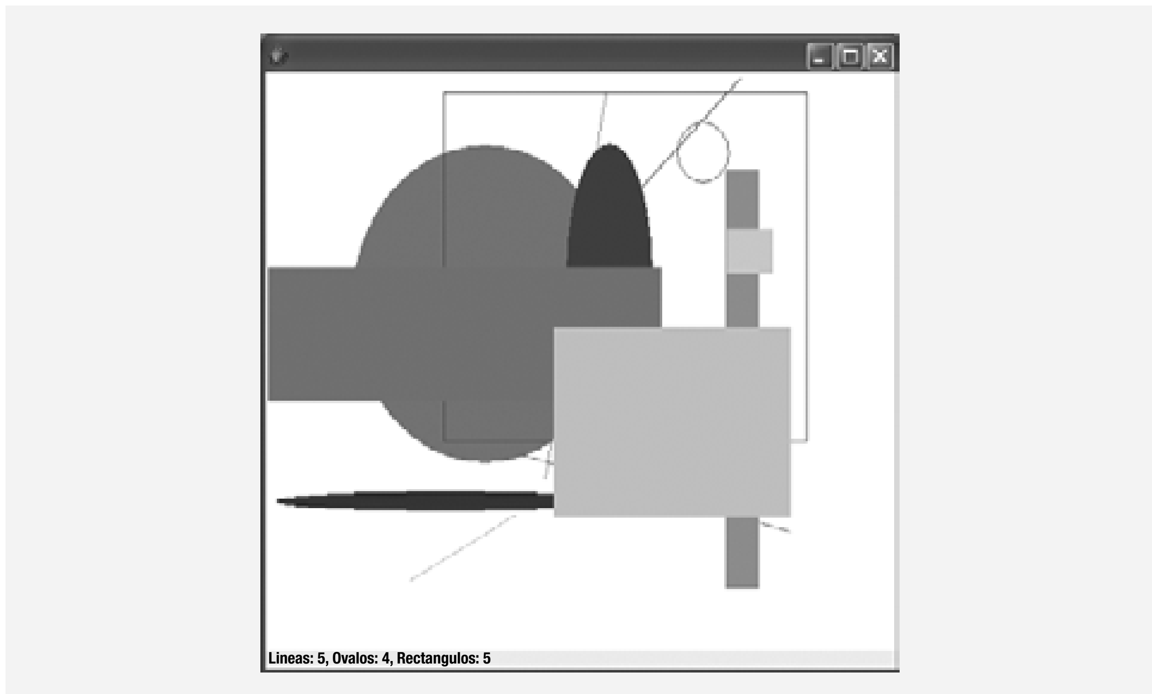


Figura 9.20 | Objeto `JLabel` que muestra las estadísticas de las figuras.

9.9 Conclusión

En este capítulo se introdujo el concepto de la herencia: la habilidad de crear clases mediante la absorción de los miembros de una clase existente, mejorándolos con nuevas capacidades. Usted aprendió las nociones de las superclases y las subclases, y utilizó la palabra clave `extends` para crear una subclase que hereda miembros de una superclase. En este capítulo se introdujo también el modificador de acceso `protected`; los métodos de la subclase pueden acceder a los miembros `protected` de la superclase. Aprendió también cómo acceder a los miembros de la superclase mediante `super`. Vio además cómo se utilizan los constructores en las jerarquías de herencia. Por último, aprendió acerca de los métodos de la clase `Object`, la superclase directa o indirecta de todas las clases en Java.

En el capítulo 10, Programación orientada a objetos: polimorfismo, continuaremos con nuestra discusión sobre la herencia al introducir el polimorfismo: un concepto orientado a objetos que nos permite escribir programas que puedan manipular convenientemente, de una forma más general, objetos de una amplia variedad de clases relacionadas por la herencia. Después de estudiar el capítulo 10, estará familiarizado con las clases, los objetos, el encapsulamiento, la herencia y el polimorfismo: las tecnologías clave de la programación orientada a objetos.

Resumen

Sección 9.1 Introducción

- La reutilización de software reduce el tiempo de desarrollo de los programas.
- La superclase directa de una subclase (que se especifica mediante la palabra `extends` en la primera línea de una declaración de clase) es la superclase a partir de la cual hereda la subclase. Una superclase indirecta de una subclase se encuentra dos o más niveles arriba de esa subclase en la jerarquía de clases.
- En la herencia simple, una clase se deriva de una superclase directa. En la herencia múltiple, una clase se deriva de más de una superclase directa. Java no soporta la herencia múltiple.
- Una subclase es más específica que su superclase, y representa un grupo más pequeño de objetos.
- Cada objeto de una subclase es también un objeto de la superclase de esa clase. Sin embargo, el objeto de una superclase no es un objeto de las subclases de su clase.
- Una relación *"es un"* representa a la herencia. En una relación *"es un"*, un objeto de una subclase también puede tratarse como un objeto de su superclase.
- Una relación *"tiene un"* representa a la composición. En una relación *"tiene un"*, el objeto de una clase contiene referencias a objetos de otras clases.

Sección 9.2 Superclases y subclases

- Las relaciones de herencia simple forman estructuras jerárquicas tipo árbol; una superclase existe en una relación jerárquica con sus subclases.

Sección 9.3 Miembros `protected`

- Los miembros `public` de una superclase son accesibles en cualquier parte en donde el programa tenga una referencia a un objeto de esa superclase, o de una de sus subclases.
- Los miembros `private` de una superclase son accesibles sólo dentro de la declaración de esa superclase.
- Los miembros `protected` de una superclase tienen un nivel intermedio de protección entre acceso `public` y `private`. Pueden ser utilizados por los miembros de la superclase, los miembros de sus subclases y los miembros de otras clases en el mismo paquete.
- Cuando un método de una subclase sobrescribe a un método de una superclase, se puede acceder al método de la superclase desde la subclase, si se antepone al nombre del método de la subclase la palabra clave `super` y un separador punto (`.`).

Sección 9.4 Relación entre las superclases y las subclases

- Una subclase no puede acceder o heredar los miembros `private` de su superclase; al permitir esto se violaría el encapsulamiento de la superclase. Sin embargo, una subclase puede heredar los miembros no `private` de su superclase.
- El método de una superclase puede sobrescribirse en una clase para declarar una implementación apropiada para la subclase.
- El método `toString` no recibe argumentos y devuelve un objeto `String`. Por lo general, una subclase sobrescribe el método `toString` de la clase `Object`.
- Cuando se imprime un objeto usando el especificador de formato `%s`, se hace una llamada implícita al método `toString` del objeto para obtener su representación de cadena.

Sección 9.5 Los constructores en las subclases

- La primera tarea de cualquier constructor de subclase es llamar al constructor de su superclase directa, ya sea en forma explícita o implícita, para asegurar que las variables de instancia heredadas de la superclase se inicialicen en forma apropiada.

- Una subclase puede invocar en forma explícita a un constructor de su superclase; para ello utiliza la sintaxis de llamada del constructor de la superclase: la palabra clave `super`, seguida de un conjunto de paréntesis que contienen los argumentos del constructor de la superclase.

Sección 9.6 Ingeniería de Software mediante la herencia

- Declarar variables de instancia `private`, al mismo tiempo que se proporcionan métodos no `private` para manipular y realizar la validación, ayuda a cumplir con la buena ingeniería de software.

Terminología

biblioteca de clases
clase base
clase derivada
`clone`, método de la clase `Object`
componentes reutilizables estandarizados
composición
constructor de subclase
constructor de superclase
constructor de superclase sin argumentos
diagrama de jerarquía
`equals`, método de la clase `Object`
es un, relación
especialización
`extends`, palabra clave
`getClass`, método de la clase `Object`
`hashCode`, método de la clase `Object`
herencia
herencia simple
invocar al constructor de una superclase
invocar al método de una superclase
jerarquía de clases
jerarquía de herencia
método heredado
miembro heredado
`Object`, clase
objeto de una subclase
objeto de una superclase
`private`, miembro de superclase
`protected`, miembro de superclase
`protected`, palabra clave
`public`, miembro de superclase
relación jerárquica
reutilización de software
sintaxis de llamada al constructor de una superclase
sobrescribir (redefinir) el método de una superclase
software frágil
software quebradizo
subclase
`super`, palabra clave
superclase
superclase directa
superclase indirecta
tiene un, relación
`toString`, método de la clase `Object`

Ejercicios de autoevaluación

9.1 Complete las siguientes oraciones:

a) _____________ es una forma de reutilización de software, en la que nuevas clases adquieren los miembros de las clases existentes, y se mejoran con nuevas capacidades.
b) Los miembros _____________ de una superclase pueden utilizarse en la declaración de la superclase y en las declaraciones de las subclases.
c) En una relación _____________, un objeto de una subclase puede ser tratado también como un objeto de su superclase.
d) En una relación _____________, el objeto de una clase tiene referencias a objetos de otras clases como miembros.
e) En la herencia simple, una clase existe en una relación _____________ con sus subclases.
f) Los miembros _____________ de una superclase son accesibles en cualquier parte en donde el programa tenga una referencia a un objeto de esa superclase, o a un objeto de una de sus subclases.
g) Cuando se crea la instancia de un objeto de una subclase, el _____________ de una superclase se llama en forma implícita o explícita.
h) Los constructores de una subclase pueden llamar a los constructores de la superclase mediante la palabra clave _____________.

9.2 Conteste con *verdadero* o *falso* a cada una de las siguientes proposiciones; en caso de ser *falso*, explique por qué.

a) Los constructores de la superclase no son heredados por las subclases.
b) Una relación *"tiene un"* se implementa mediante la herencia.

c) Una clase `Auto` tiene una relación *"es un"* con las clases `VolanteDireccion` y `Frenos`.
d) La herencia fomenta la reutilización de software comprobado, de alta calidad.
e) Cuando una subclase redefine al método de una superclase utilizando la misma firma, se dice que la subclase sobrecarga a ese método de la superclase.

Respuestas a los ejercicios de autoevaluación

9.1 a) Herencia. b) `public` y `protected`. c) *"es un"* o de herencia. d) *"tiene-un"*, o composición. e) jerárquica. f) `public`. g) constructor. h) `super`.

9.2 a) Verdadero. b) Falso. Una relación *"tiene un"* se implementa mediante la composición. Una relación *"es-un"* se implementa mediante la herencia. c) Falso. Éste es un ejemplo de una relación *"tiene un"*. La clase `Auto` tiene una relación *"es-un"* con la clase `Vehiculo`. d) Verdadero. e) Falso. Esto se conoce como sobrescritura, no sobrecarga; un método sobrecargado tiene el mismo nombre, pero una firma distinta.

Ejercicios

9.3 Muchos programas escritos con herencia podrían escribirse mediante la composición, y viceversa. Vuelva a escribir las clases `EmpleadoBaseMasComision4` (figura 9.13) de la jerarquía `EmpleadoPorComision3-EmpleadoBaseMasComision4` para usar la composición en vez de la herencia. Una vez que haga esto, valore los méritos relativos de las dos metodologías para los problemas de `EmpleadoPorComision3` y `EmpleadoBaseMasComision4`, así como también para los programas orientados a objetos en general. ¿Cuál metodología es más natural? ¿Por qué?

9.4 Describa las formas en las que la herencia fomenta la reutilización de software, ahorra tiempo durante el desarrollo de los programas y ayuda a prevenir errores.

9.5 Dibuje una jerarquía de herencia para los estudiantes en una universidad, de manera similar a la jerarquía que se muestra en la figura 9.2. Use a `Estudiante` como la superclase de la jerarquía, y después extienda `Estudiante` con las clases `EstudianteNoGraduado` y `EstudianteGraduado`. Siga extendiendo la jerarquía con el mayor número de niveles que sea posible. Por ejemplo, `EstudiantePrimerAnio`, `EstudianteSegundoAnio`, `EstudianteTercerAnio` y `EstudianteCuartoAnio` podrían extender a `EstudianteNoGraduado`, y `EstudianteDoctorado` y `EstudianteMaestria` podrían ser subclases de `EstudianteGraduado`. Después de dibujar la jerarquía, hable sobre las relaciones que existen entre las clases. [*Nota:* no necesita escribir código para este ejercicio].

9.6 El mundo de las figuras es más extenso que las figuras incluidas en la jerarquía de herencia de la figura 9.3. Anote todas las figuras en las que pueda pensar (tanto bidimensionales como tridimensionales) e intégrelas en una jerarquía `Figura` más completa, con todos los niveles que sea posible. Su jerarquía debe tener la clase `Figura` en la parte superior. Las clases `FiguraBidimensional` y `FiguraTridimensional` deben extender a `Figura`. Agregue subclases adicionales, como `Cuadrilatero` y `Esfera`, en sus ubicaciones correctas en la jerarquía, según sea necesario.

9.7 Algunos programadores prefieren no utilizar el acceso `protected`, pues piensan que quebranta el encapsulamiento de la superclase. Hable sobre los méritos relativos de utilizar el acceso `protected`, en comparación con el acceso `private` en las superclases.

9.8 Escriba una jerarquía de herencia para las clases `Cuadrilatero`, `Trapezoide`, `Paralelogramo`, `Rectangulo` y `Cuadrado`. Use `Cuadrilatero` como la superclase de la jerarquía. Agregue todos los niveles que sea posible a la jerarquía. Especifique las variables de instancia y los métodos para cada clase. Las variables de instancia `private` de `Cuadrilatero` deben ser los pares de coordenadas *x-y* para los cuatro puntos finales del `Cuadrilatero`. Escriba un programa que cree instancias de objetos de sus clases, y que imprima el área de cada objeto (excepto `Cuadrilatero`).

10

Programación orientada a objetos: polimorfismo



Un anillo para gobernarlos a todos, un anillo para encontrarlos,
un anillo para traerlos a todos y en la oscuridad enlazarlos.

—John Ronald Reuel Tolkien

Las proposiciones generales no deciden casos concretos.

—Oliver Wendell Holmes

Un filósofo de imponente estatura no piensa en un vacío.
Incluso sus ideas más abstractas son, en cierta medida, condicionadas por lo que se conoce o no en el tiempo en que vive.

—Alfred North Whitehead

¿Por qué, alma mía, desfalleces
y te agitas por mí?

—Salmos 42:5

OBJETIVOS

En este capítulo aprenderá a:

- Comprender el concepto de polimorfismo.
- Aprender a utilizar métodos sobrescritos para llevar a cabo el polimorfismo.
- Distinguir entre clases abstractas y concretas.
- Aprender a declarar métodos `abstract` para crear clases abstractas.
- Apreciar la manera en que el polimorfismo hace que los sistemas puedan extenderse y mantenerse.
- Determinar el tipo de un objeto en tiempo de ejecución.
- Aprender a declarar e implementar interfaces.

Plan general

10.1 Introducción

Ahora continuaremos nuestro estudio de la programación orientada a objetos, explicando y demostrando el **polimorfismo** con las jerarquías de herencia. El polimorfismo nos permite "programar en forma general", en vez de "programar en forma específica". En especial, nos permite escribir programas que procesen objetos que compartan la misma superclase en una jerarquía de clases, como si todos fueran objetos de la superclase; esto puede simplificar la programación.

Considere el siguiente ejemplo de polimorfismo. Suponga que crearemos un programa que simula el movimiento de varios tipos de animales para un estudio biológico. Las clases `Pez`, `Rana` y `Ave` representan los tres tipos de animales bajo investigación. Imagine que cada una de estas clases extiende a la superclase `Animal`, la cual contiene un método llamado `mover` y mantiene la posición actual de un animal, en forma de coordenadas *x-y*. Cada subclase implementa el método `mover`. Nuestro programa mantiene un arreglo de referencias a objetos de las diversas subclases de `Animal`. Para simular los movimientos de los animales, el programa envía a cada objeto el mismo mensaje una vez por segundo; a saber, `mover`. No obstante, cada tipo específico de `Animal` responde a un mensaje `mover` de manera única; un `Pez` podría nadar tres pies, una `Rana` podría saltar cinco pies y un `Ave` podría volar diez pies. El programa envía el mismo mensaje (es decir, `mover`) a cada objeto animal en forma genérica, pero cada objeto sabe cómo modificar sus coordenadas *x-y* en forma apropiada para su tipo específico de movimiento. Confiar en que cada objeto sepa cómo "hacer lo correcto" (es decir, lo que sea apropiado para ese tipo de objeto) en respuesta a la llamada al mismo método es el concepto clave del polimorfismo. El mismo mensaje

(en este caso, `mover`) que se envía a una variedad de objetos tiene "muchas formas" de resultados; de aquí que se utilice el término polimorfismo.

Con el polimorfismo podemos diseñar e implementar sistemas que puedan extenderse con facilidad; pueden agregarse nuevas clases con sólo modificar un poco (o nada) las porciones generales de la aplicación, siempre y cuando las nuevas clases sean parte de la jerarquía de herencia que la aplicación procesa en forma genérica. Las únicas partes de un programa que deben alterarse para dar cabida a las nuevas clases son las que requieren un conocimiento directo de las nuevas clases que el programador agregará a la jerarquía. Por ejemplo, si extendemos la clase `Animal` para crear la clase `Tortuga` (que podría responder a un mensaje `mover` caminando una pulgada), necesitamos escribir sólo la clase `Tortuga` y la parte de la simulación que crea una instancia de un objeto `Tortuga`. Las porciones de la simulación que procesan a cada `Animal` en forma genérica pueden permanecer iguales.

Este capítulo se divide en varias partes. Primero hablaremos sobre los ejemplos comunes del polimorfismo. Después proporcionaremos un ejemplo que demuestra el comportamiento polimórfico. Utilizaremos referencias a la superclase para manipular tanto a los objetos de la superclase como a los objetos de las subclases mediante el polimorfismo.

Después presentaremos un ejemplo práctico en el que utilizaremos nuevamente la jerarquía de empleados de la sección 9.4.5. Desarrollaremos una aplicación simple de nómina que calcula mediante el polimorfismo el salario semanal de varios tipos de empleados, usando el método `ingresos` de cada empleado. Aunque los ingresos de cada tipo de empleado se calculan de una manera específica, el polimorfismo nos permite procesar a los empleados "en general". En el ejemplo práctico ampliaremos la jerarquía para incluir dos nuevas clases: `EmpleadoAsalariado` (para las personas que reciben un salario semanal fijo) y `EmpleadoPorHoras` (para las personas que reciben un salario por horas y "tiempo y medio" por el tiempo extra). Declararemos un conjunto común de funcionalidad para todas las clases en la jerarquía actualizada en una clase "abstracta" llamada `Empleado`, a partir de la cual las clases `EmpleadoAsalariado`, `EmpleadoPorHoras` y `EmpleadoPorComision` heredan en forma directa, y la clase `EmpleadoBaseMasComision4` hereda en forma indirecta. Como pronto verá, al invocar el método `ingresos` de cada empleado desde una referencia a la superclase `Empleado`, se realiza el cálculo correcto de los ingresos gracias a las capacidades polimórficas de Java.

Algunas veces, cuando se lleva a cabo el procesamiento polimórfico, es necesario programar "en forma específica". Nuestro ejemplo práctico con `Empleado` demuestra que un programa puede determinar el tipo de un objeto en tiempo de ejecución, y actuar sobre ese objeto de manera acorde. En el ejemplo práctico utilizamos estas capacidades para determinar si cierto objeto empleado específico *es un* `EmpleadoBaseMasComision`. Si es así, incrementamos el salario base de ese empleado en un 10%.

El capítulo continúa con una introducción a las interfaces en Java. Una interfaz describe a un conjunto de métodos que pueden llamarse en un objeto, pero no proporciona implementaciones concretas para ellos. Los programadores pueden declarar clases que **implementen** a (es decir, que proporcionen implementaciones concretas para los métodos de) una o más interfaces. Cada método de una interfaz debe declararse en todas las clases que implementen a la interfaz. Una vez que una clase implementa a una interfaz, todos los objetos de esa clase tienen una relación *"es un"* con el tipo de la interfaz, y se garantiza que todos los objetos de la clase proporcionarán la funcionalidad descrita por la interfaz. Esto se aplica también para todas las subclases de esa clase.

En especial, las interfaces son útiles para asignar la funcionalidad común a clases que posiblemente no estén relacionadas. Esto permite que los objetos de clases no relacionadas se procesen en forma polimórfica; los objetos de las clases que implementan la misma interfaz pueden responder a las mismas llamadas a los métodos. Para demostrar la creación y el uso de interfaces, modificaremos nuestra aplicación de nómina para crear una aplicación general de cuentas por pagar, que puede calcular los pagos vencidos por los ingresos de los empleados de la compañía y los montos de las facturas a pagar por los bienes comprados. Como verá, las interfaces permiten capacidades polimórficas similares a las que permite la herencia.

10.2 Ejemplos del polimorfismo

Ahora consideraremos diversos ejemplos adicionales. Si la clase `Rectangulo` se deriva de la clase `Cuadrilatero`, entonces un objeto `Rectangulo` es una versión más específica de un objeto `Cuadrilatero`. Cualquier operación (por ejemplo, calcular el perímetro o el área) que pueda realizarse en un objeto `Cuadrilatero` también puede realizarse en un objeto `Rectangulo`. Estas operaciones también pueden realizarse en otros objetos `Cuadrilatero`, como `Cuadrado`, `Paralelogramo` y `Trapezoide`. El polimorfismo ocurre cuando un programa invoca a un método a través de una variable de la superclase; en tiempo de ejecución, se hace una llamada a la versión correcta

del método de la subclase, con base en el tipo de la referencia almacenada en la variable de la superclase. En la sección 10.3 veremos un ejemplo de código simple, en el cual se ilustra este proceso.

Como otro ejemplo, suponga que diseñaremos un videojuego que manipule objetos de las clases `Marciano`, `Venusino`, `Plutoniano`, `NaveEspacial` y `RayoLaser`. Imagine que cada clase hereda de la superclase común llamada `ObjetoEspacial`, la cual contiene el método `dibujar`. Cada subclase implementa a este método. Un programa administrador de la pantalla mantiene una colección (por ejemplo, un arreglo `ObjetoEspacial`) de referencias a objetos de las diversas clases. Para refrescar la pantalla, el administrador de pantalla envía en forma periódica el mismo mensaje a cada objeto; a saber, `dibujar`. No obstante, cada objeto responde de una manera única. Por ejemplo, un objeto `Marciano` podría dibujarse a sí mismo en color rojo, con ojos verdes y el número apropiado de antenas. Un objeto `NaveEspacial` podría dibujarse a sí mismo como un platillo volador de color plata brillante; un objeto `RayoLaser`, como un rayo color rojo brillante a lo largo de la pantalla. De nuevo, el mismo mensaje (en este caso, `dibujar`) que se envía a una variedad de objetos tiene "muchas formas" de resultados.

Un administrador de pantalla polimórfico podría utilizar el polimorfismo para facilitar el proceso de agregar nuevas clases a un sistema, con el menor número de modificaciones al código del sistema. Suponga que deseamos agregar objetos `Mercuriano` a nuestro videojuego. Para ello, debemos crear una clase `Mercuriano` que extienda a `ObjetoEspacial` y proporcione su propia implementación del método `dibujar`. Cuando aparezcan objetos de la clase `Mercuriano` en la colección `ObjetoEspacial`, el código del administrador de pantalla invocará al método `dibujar`, de la misma forma que para cualquier otro objeto en la colección, sin importar su tipo. Por lo tanto, los nuevos objetos `Mercuriano` simplemente se integran al videojuego sin necesidad de que el programador modifique el código del administrador de pantalla. Así, sin modificar el sistema (más que para crear nuevas clases y modificar el código que genera nuevos objetos), los programadores pueden utilizar el polimorfismo para incluir de manera conveniente tipos adicionales que no se hayan considerado a la hora de crear el sistema.

Con el polimorfismo podemos usar el mismo nombre y la misma firma del método para hacer que ocurran distintas acciones, dependiendo del tipo del objeto en el que se invoca el método. Esto proporciona al programador una enorme capacidad expresiva.



Observación de ingeniería de software 10.1

El polimorfismo permite a los programadores tratar con las generalidades y dejar que el entorno en tiempo de ejecución se encargue de los detalles específicos. Los programadores pueden ordenar a los objetos que se comporten en formas apropiadas para ellos, sin necesidad de conocer los tipos de los objetos (siempre y cuando éstos pertenezcan a la misma jerarquía de herencia).



Observación de ingeniería de software 10.2

El polimorfismo promueve la extensibilidad: el software que invoca el comportamiento polimórfico es independiente de los tipos de los objetos a los cuales se envían los mensajes. Se pueden incorporar en un sistema nuevos tipos de objetos que pueden responder a las llamadas de los métodos existentes, sin necesidad de modificar el sistema base. Sólo el código cliente que crea instancias de los nuevos objetos debe modificarse para dar cabida a los nuevos tipos.

10.3 Demostración del comportamiento polimórfico

En la sección 9.4 creamos una jerarquía de clases de empleados por comisión, en la cual la clase `EmpleadoBaseMasComision` heredó de la clase `EmpleadoPorComision`. Los ejemplos en esa sección manipularon objetos `EmpleadoPorComision` y `EmpleadoBaseMasComision` mediante el uso de referencias a ellos para invocar a sus métodos; dirigimos las referencias a la superclase a los objetos de la superclase, y las referencias a la subclase a los objetos de la subclase. Estas asignaciones son naturales y directas; las referencias a la superclase están diseñadas para referirse a objetos de la superclase, y las referencias a la subclase están diseñadas para referirse a objetos de la subclase. No obstante, como veremos pronto, es posible realizar otras asignaciones.

En el siguiente ejemplo, dirigiremos una referencia a la superclase a un objeto de la subclase. Después mostraremos cómo al invocar un método en un objeto de la subclase a través de una referencia a la superclase se invoca a la funcionalidad de la subclase; el tipo del *objeto actual al que se hace referencia*, no el tipo de *referencia*, es el que determina cuál método se llamará. Este ejemplo demuestra el concepto clave de que un objeto de una subclase puede tratarse como un objeto de su superclase. Esto permite varias manipulaciones interesantes. Un programa puede crear un arreglo de referencias a la superclase, que se refieran a objetos de muchos tipos de sub-

clases. Esto se permite, ya que cada objeto de una subclase *es un* objeto de su superclase. Por ejemplo, podemos asignar la referencia de un objeto `EmpleadoBaseMasComision` a una variable de la superclase `EmpleadoPorComision`, ya que un `EmpleadoBaseMasComision` *es un* `EmpleadoPorComision`; por lo tanto, podemos tratar a un `EmpleadoBaseMasComision` como un `EmpleadoPorComision`.

Como veremos más adelante en este capítulo, no podemos tratar a un objeto de la superclase como un objeto de cualquiera de sus subclases, porque un objeto superclase no es un objeto de ninguna de sus subclases. Por ejemplo, no podemos asignar la referencia de un objeto `EmpleadoPorComision` a una variable de la subclase `EmpleadoBaseMasComision`, ya que un `EmpleadoPorComision` no es un `EmpleadoBaseMasComision`, no tiene una variable de instancia `salarioBase` y no tiene los métodos `establecerSalarioBase` y `obtenerSalarioBase`. La relación *"es un"* se aplica sólo de una subclase a sus superclases directas (e indirectas), pero no viceversa.

El compilador de Java permite asignar una referencia a la superclase a una variable de la subclase, si convertimos explícitamente la referencia a la superclase al tipo de la subclase; una técnica que veremos con más detalle en la sección 10.5. ¿Para qué nos serviría, en un momento dado, realizar una asignación así? Una referencia a la superclase puede usarse para invocar sólo a los métodos declarados en la superclase; si tratamos de invocar métodos que sólo pertenezcan a la subclase, a través de una referencia a la superclase, se producen errores de compilación. Si un programa necesita realizar una operación específica para la subclase en un objeto de la subclase al que se haga una referencia mediante una variable de la superclase, el programa primero debe convertir la referencia a la superclase en una referencia a la subclase, mediante una técnica conocida como **conversión descendente**. Esto permite al programa invocar métodos de la subclase que no se encuentren en la superclase. En la sección 10.5 presentaremos un ejemplo concreto de conversión descendente.

El ejemplo de la figura 10.1 demuestra tres formas de usar variables de la superclase y la subclase para almacenar referencias a objetos de la superclase y de la subclase. Las primeras dos formas son simples: al igual que en la sección 9.4, asignamos una referencia a la superclase a una variable de la superclase, y asignamos una referencia a la subclase a una variable de la subclase. Después demostramos la relación entre las subclases y las superclases (es decir, la relación *"es-un"*) mediante la asignación de una referencia a la subclase a una variable de la superclase. [*Nota:* este programa utiliza las clases `EmpleadoPorComision3` y `EmpleadoBaseMasComision4` de las figuras 9.12 y 9.13, respectivamente].

```
// Fig. 10.1: PruebaPolimorfismo.java
// Asignación de referencias a la superclase y la subclase, a
// variables de la superclase y la subclase.

public class PruebaPolimorfismo
{
   public static void main( String args[] )
   {
      // asigna la referencia a la superclase a una variable de la superclase
      EmpleadoPorComision3 empleadoPorComision = new EmpleadoPorComision3(
         "Sue", "Jones", "222-22-2222", 10000, .06 );

      // asigna la referencia a la subclase a una variable de la subclase
      EmpleadoBaseMasComision4 empleadoBaseMasComision =
         new EmpleadoBaseMasComision4(
         "Bob", "Lewis", "333-33-3333", 5000, .04, 300 );

      // invoca a toString en un objeto de la superclase, usando una variable de la
      superclase
      System.out.printf( "%s %s:\n\n%s\n\n",
         "Llamada a toString de EmpleadoPorComision3 con referencia de superclase ",
         "a un objeto de la superclase", empleadoPorComision.toString() );

      // invoca a toString en un objeto de la subclase, usando una variable de la
      subclase
```

Figura 10.1 | Asignación de referencias a la superclase y la subclase, a variables de la superclase y la subclase. (Parte 1 de 2).

```
24         System.out.printf( "%s %s:\n\n%s\n\n",
25            "Llamada a toString de EmpleadoBaseMasComision4 con referencia",
26            "de subclase a un objeto de la subclase"
27            empleadoBaseMasComision.toString() );
28
29         // invoca a toString en un objeto de la subclase, usando una variable de la
           superclase
30         EmpleadoPorComision3 empleadoPorComision2 =
31            empleadoBaseMasComision;
32         System.out.printf( "%s %s:\n\n%s\n",
33            "Llamada a toString de EmpleadoBaseMasComision4 con referencia de superclase",
34            "a un objeto de la subclase", empleadoPorComision2.toString() );
35      } // fin de main
36   } // fin de la clase PruebaPolimorfismo
```

```
Llamada a toString de EmpleadoPorComision3 con referencia de superclase a un objeto de la
superclase:

empleado por comision: Sue Jones
numero de seguro social: 222-22-2222
ventas brutas: 10000.00
tarifa de comision: 0.06

Llamada a toString de EmpleadoBaseMasComision4 con referencia de subclase a un objeto de la
subclase:

con sueldo base empleado por comision: Bob Lewis
numero de seguro social: 333-33-3333
ventas brutas: 5000.00
tarifa de comision: 0.04
sueldo base: 300.00

Llamada a toString de EmpleadoBaseMasComision4 con referencia de superclase a un objeto de
la subclase:

con sueldo base empleado por comision: Bob Lewis
numero de seguro social: 333-33-3333
ventas brutas: 5000.00
tarifa de comision: 0.04
sueldo base: 300.00
```

Figura 10.1 | Asignación de referencias a la superclase y la subclase, a variables de la superclase y la subclase. (Parte 2 de 2).

En la figura 10.1, las líneas 10 y 11 crean un objeto `EmpleadoPorComision3` y asignan su referencia a una variable `EmpleadoPorComision3`. Las líneas 14 a 16 crean un objeto `EmpleadoBaseMasComision4` y asignan su referencia a una variable `EmpleadoBaseMasComision4`. Estas asignaciones son naturales; por ejemplo, el principal propósito de una variable `EmpleadoPorComision3` es guardar una referencia a un objeto `EmpleadoPorComision3`. Las líneas 19 a 21 utilizan la referencia `empleadoPorComision` para invocar a `toString` en forma explícita. Como `empleadoPorComision` hace referencia a un objeto `EmpleadoPorComision3`, se hace una llamada a la versión de `toString` de la superclase `EmpleadoPorComision3`. De manera similar, las líneas 24 a 27 utilizan a `empleadoBaseMasComision` para invocar a `toString` de forma explícita en el objeto `EmpleadoBaseMasComision4`. Esto invoca a la versión de `toString` de la subclase `EmpleadoBaseMasComision4`.

Después, las líneas 30 y 31 asignan la referencia al objeto `empleadoBaseMasComision` de la subclase a una variable de la superclase `EmpleadoPorComision3`, que las líneas 32 a 34 utilizan para invocar al método `toString`. Cuando una variable de la superclase contiene una referencia a un objeto de la subclase, y esta referencia se utiliza para llamar a un método, se hace una llamada a la versión del método de la subclase. Por ende, `empleadoPorComision2.ToString()` en la línea 34 en realidad llama al método `toString` de la clase

EmpleadoBaseMasComision4. El compilador de Java permite este "cruzamiento", ya que un objeto de una subclase *es un* objeto de su superclase (pero no viceversa). Cuando el compilador encuentra una llamada a un método que se realiza a través de una variable, determina si el método puede llamarse verificando el tipo de clase de la variable. Si esa clase contiene la declaración del método apropiada (o hereda una), se compila la llamada. En tiempo de ejecución, el tipo del objeto al cual se refiere la variable es el que determina el método que se utilizará.

10.4 Clases y métodos abstractos

Cuando pensamos en un tipo de clase, asumimos que los programas crearán objetos de ese tipo. No obstante, en algunos casos es conveniente declarar clases para las cuales el programador nunca creará instancias de objetos. A dichas clases se les conoce como **clases abstractas**. Como se utilizan sólo como superclases en jerarquías de herencia, nos referimos a ellas como **superclases abstractas**. Estas clases no pueden utilizarse para instanciar objetos, ya que como veremos pronto, las clases abstractas están incompletas. Las subclases deben declarar las "piezas faltantes". En la sección 10.5 demostraremos las clases abstractas.

El propósito de una clase abstracta es proporcionar una superclase apropiada, a partir de la cual puedan heredar otras clases y, por ende, compartir un diseño común. Por ejemplo, en la jerarquía de Figura de la figura 9.3, las subclases heredan la noción de lo que significa ser una Figura; los atributos comunes como posicion, color y grosorBorde, y los comportamientos como dibujar, mover, cambiarTamanio y cambiarColor. Las clases que pueden utilizarse para instanciar objetos se llaman **clases concretas**. Dichas clases proporcionan implementaciones de cada método que declaran (algunas de las implementaciones pueden heredarse). Por ejemplo, podríamos derivar las clases concretas Circulo, Cuadrado y Triangulo de la superclase abstracta FiguraBidimensional. De manera similar, podríamos derivar las clases concretas Esfera, Cubo y Tetraedro de la superclase abstracta FiguraTridimensional. Las superclases abstractas son demasiado generales como para crear objetos reales; sólo especifican lo que tienen en común las subclases. Necesitamos ser más específicos para poder crear objetos. Por ejemplo, si envía el mensaje dibujar a la clase abstracta FiguraBidimensional, la clase sabe que las figuras bidimensionales deben poder dibujarse, pero no sabe qué figura específica dibujar, por lo que no puede implementar un verdadero método dibujar. Las clases concretas proporcionan los detalles específicos que hacen razonable la creación de instancias de objetos.

No todas las jerarquías de herencia contienen clases abstractas. Sin embargo, a menudo los programadores escriben código cliente que utiliza sólo tipos de superclases abstractas para reducir las dependencias del código cliente en un rango de tipos de subclases específicas. Por ejemplo, un programador puede escribir un método con un parámetro de un tipo de superclase abstracta. Cuando se llama, ese método puede recibir un objeto de cualquier clase concreta que extienda en forma directa o indirecta a la superclase especificada como el tipo del parámetro.

Algunas veces las clases abstractas constituyen varios niveles de la jerarquía. Por ejemplo, la jerarquía de Figura de la figura 9.3 empieza con la clase abstracta Figura. En el siguiente nivel de la jerarquía hay dos clases abstractas más, FiguraBidimensional y FiguraTridimensional. El siguiente nivel de la jerarquía declara clases concretas para objetos FiguraBidimensional (Circulo, Cuadrado y Triangulo) y para objetos FiguraTridimensional (Esfera, Cubo y Tetraedro).

Para hacer una clase abstracta, ésta se declara con la palabra clave **abstract**. Por lo general, una clase abstracta contiene uno o más **métodos abstractos**. Un método abstracto tiene la palabra clave abstract en su declaración, como en

```
public abstract void dibujar(); // método abstracto
```

Los métodos abstractos no proporcionan implementaciones. Una clase que contiene métodos abstractos debe declararse como clase abstracta, aun si esa clase contiene métodos concretos (no abstractos). Cada subclase concreta de una superclase abstracta también debe proporcionar implementaciones concretas de los métodos abstractos de la superclase. Los constructores y los métodos static no pueden declararse como abstract. Los constructores no se heredan, por lo que nunca podría implementarse un constructor abstract. Aunque los métodos static se heredan, no están asociados con objetos específicos de las clases que los declaran. Como el propósito de los métodos abstract es sobrescribirlos para procesar objetos con base en sus tipos, no tendría sentido declarar un método static como abstract.



Observación de ingeniería de software 10.3

Una clase abstracta declara los atributos y comportamientos comunes de las diversas clases en una jerarquía de clases. Por lo general, una clase abstracta contiene uno o más métodos abstractos, que las subclases deben sobrescribir, si van a ser concretas. Las variables de instancia y los métodos concretos de una clase abstracta están sujetos a las reglas normales de la herencia,



Error común de programación 10.1

Tratar de instanciar un objeto de una clase abstracta es un error de compilación.



Error común de programación 10.2

Si no se implementan los métodos abstractos de la superclase en una clase derivada, se produce un error de compilación, a menos que la clase derivada también se declare como `abstract`.

Aunque no podemos instanciar objetos de superclases abstractas, pronto veremos que *podemos* usar superclases abstractas para declarar variables que puedan guardar referencias a objetos de cualquier clase concreta que se derive de esas superclases abstractas. Por lo general, los programas utilizan dichas variables para manipular los objetos de las subclases mediante el polimorfismo. Además, podemos usar los nombres de las superclases abstractas para invocar métodos `static` que estén declarados en esas superclases abstractas.

Considere otra aplicación del polimorfismo. Un programa de dibujo necesita mostrar en pantalla muchas figuras, incluyendo nuevos tipos de figuras que el programador agregará al sistema después de escribir el programa de dibujo. Este programa podría necesitar mostrar figuras, como `Circulos`, `Triangulos`, `Rectangulos` u otras, que se deriven de la superclase abstracta `Figura`. El programa de dibujo utiliza variables de `Figura` para administrar los objetos que se muestran en pantalla. Para dibujar cualquier objeto en esta jerarquía de herencia, el programa de dibujo utiliza una variable de la superclase `Figura` que contiene una referencia al objeto de la subclase para invocar al método `dibujar` del objeto. Este método se declara como `abstract` en la superclase `Figura`, por lo que cada subclase concreta *debe* implementar el método `dibujar` en una forma que sea específica para esa figura. Cada objeto en la jerarquía de herencia de `Figura` sabe cómo dibujarse a sí mismo. El programa de dibujo no tiene que preocuparse acerca del tipo de cada objeto, o si ha encontrado objetos de ese tipo.

En especial, el polimorfismo es efectivo para implementar los denominados sistemas de software en capas. Por ejemplo, en los sistemas operativos cada tipo de dispositivo físico puede operar en forma muy distinta a los demás. Aun así, los comandos para leer o escribir datos desde y hacia los dispositivos pueden tener cierta uniformidad. Para cada dispositivo, el sistema operativo utiliza una pieza de software llamada controlador de dispositivos para controlar toda la comunicación entre el sistema y el dispositivo. El mensaje de escritura que se envía a un objeto controlador de dispositivo necesita interpretarse de manera específica en el contexto de ese controlador, y la forma en que manipula a un dispositivo de un tipo específico. No obstante, la llamada de escritura en sí no es distinta a la escritura en cualquier otro dispositivo en el sistema: colocar cierto número de bytes de memoria en ese dispositivo. Un sistema operativo orientado a objetos podría usar una superclase abstracta para proporcionar una "interfaz" apropiada para todos los controladores de dispositivos. Después, a través de la herencia de esa superclase abstracta, se forman clases derivadas que se comporten todas de manera similar. Los métodos del controlador de dispositivos se declaran como métodos abstractos en la superclase `abstract`. Las implementaciones de estos métodos abstractos se proporcionan en las subclases que corresponden a los tipos específicos de controladores de dispositivos. Siempre se están desarrollando nuevos dispositivos, a menudo mucho después de que se ha liberado el sistema operativo. Cuando usted compra un nuevo dispositivo, éste incluye un controlador de dispositivo proporcionado por el distribuidor. El dispositivo opera de inmediato, una vez que usted lo conecta a la computadora e instala el controlador de dispositivo. Éste es otro elegante ejemplo acerca de cómo el polimorfismo hace que los sistemas sean extensibles.

En la programación orientada a objetos es común declarar una **clase iteradora** que pueda recorrer todos los objetos en una colección, como un arreglo (capítulo 7) o un objeto `ArrayList` (capítulo 19, Colecciones). Por ejemplo, un programa puede imprimir un arreglo `ArrayList` de objetos creando un objeto iterador, y luego usándolo para obtener el siguiente elemento de la lista cada vez que se llame al iterador. Los iteradores se utilizan comúnmente en la programación polimórfica para recorrer una colección que contiene referencias a objetos de diversos niveles de una jerarquía. (El capítulo 19 presenta un tratamiento detallado de `ArrayList`, los iteradores y las capacidades "genéricas"). Por ejemplo, un arreglo `ArrayList` de objetos de la clase `FiguraBidimensional`

podría contener objetos de las subclases `Cuadrado`, `Circulo`, `Triangulo` y así, sucesivamente. Al llamar al método `dibujar` para cada objeto `FiguraBidimensional` mediante una variable `FiguraBidimensional`, se dibujaría en forma polimórfica a cada objeto correctamente en la pantalla.

10.5 Ejemplo práctico: sistema de nómina utilizando polimorfismo

En esta sección analizamos de nuevo la jerarquía `EmpleadoPorComision-EmpleadoBaseMasComision` que exploramos a lo largo de la sección 9.4. Ahora podemos usar un método abstracto y polimorfismo para realizar cálculos de nómina, con base en el tipo de empleado. Crearemos una jerarquía de empleados mejorada para resolver el siguiente problema:

> *Una compañía paga a sus empleados por semana. Los empleados son de cuatro tipos: empleados asalariados que reciben un salario semanal fijo, sin importar el número de horas trabajadas; empleados por horas, que reciben un sueldo por hora y pago por tiempo extra, por todas las horas trabajadas que excedan a 40 horas; empleados por comisión, que reciben un porcentaje de sus ventas y empleados asalariados por comisión, que reciben un salario base más un porcentaje de sus ventas. Para este periodo de pago, la compañía ha decidido recompensar a los empleados asalariados por comisión, agregando un 10% a sus salarios base. La compañía desea implementar una aplicación en Java que realice sus cálculos de nómina en forma polimórfica.*

Utilizaremos la clase `abstract Empleado` para representar el concepto general de un empleado. Las clases que extienden a `Empleado` son `EmpleadoAsalariado`, `EmpleadoPorComision` y `EmpleadoPorHoras`. La clase `EmpleadoBaseMasComision` (que extiende a `EmpleadoPorComision`) representa el último tipo de empleado. El diagrama de clases de UML en la figura 10.2 muestra la jerarquía de herencia para nuestra aplicación polimórfica de nómina de empleados. Observe que la clase abstracta `Empleado` está en cursivas, según la convención de UML.

La superclase abstracta `Empleado` declara la “interfaz” para la jerarquía; esto es, el conjunto de métodos que puede invocar un programa en todos los objetos `Empleado`. Aquí utilizamos el término “interfaz” en un sentido general, para referirnos a las diversas formas en que los programas pueden comunicarse con los objetos de cualquier subclase de `Empleado`. Tenga cuidado de no confundir la noción general de una “interfaz” con la noción formal de una interfaz en Java, el tema de la sección 10.7. Cada empleado, sin importar la manera en que se calculen sus ingresos, tiene un primer nombre, un apellido paterno y un número de seguro social, por lo que las variables de instancia `private primerNombre`, `apellidoPaterno` y `numeroSeguroSocial` aparecen en la superclase abstracta `Empleado`.



Observación de ingeniería de software 10.4

Una subclase puede heredar la “interfaz” o “implementación” de una superclase. Las jerarquías diseñadas para la ***herencia de implementación*** *tienden a tener su funcionalidad en niveles altos de la jerarquía; cada nueva subclase hereda uno o más métodos que se implementaron en una superclase, y la subclase utiliza las implementaciones de la superclase. Las jerarquías diseñadas para la* ***herencia de interfaz*** *tienden a tener su funcionalidad en niveles bajos de la jerarquía; una superclase especifica uno o más métodos abstractos que deben declararse para cada clase concreta en la jerarquía, y las subclases individuales sobrescriben estos métodos para proporcionar la implementación específica para cada subclase.*

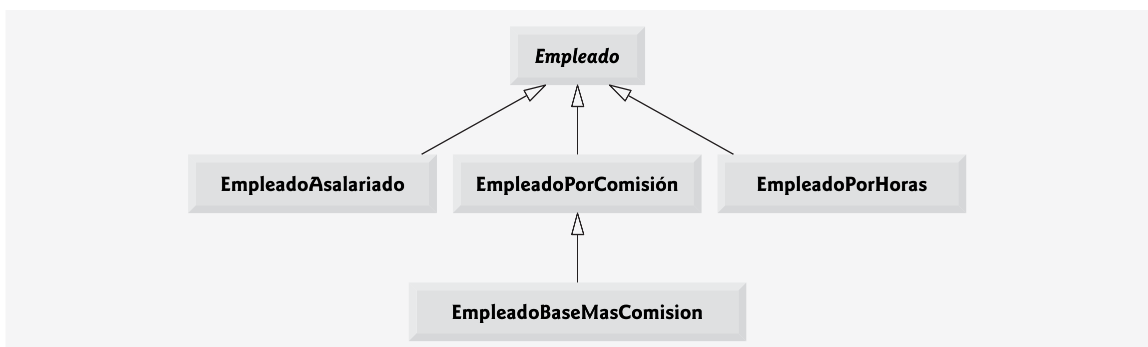


Figura 10.2 | Diagrama de clases de UML para la jerarquía de `Empleado`.

Las siguientes secciones implementan la jerarquía de clases de Empleado. Cada una de las primeras cuatro secciones implementa una de las clases concretas. La última sección implementa un programa de prueba que crea objetos de todas estas clases y procesa esos objetos mediante el polimorfismo.

10.5.1 Creación de la superclase abstracta Empleado

La clase Empleado (figura 10.4) proporciona los métodos ingresos y toString, además de los métodos *obtener* y *establecer* que manipulan las variables de instancia de Empleado. Es evidente que un método ingresos se aplica en forma genérica a todos los empleados. Pero cada cálculo de los ingresos depende de la clase de empleado. Por lo tanto, declaramos a ingresos como abstract en la superclase Empleado, ya que una implementación predeterminada no tiene sentido para ese método; no hay suficiente información para determinar qué monto debe devolver ingresos. Cada una de las subclases redefine a ingresos con una implementación apropiada. Para calcular los ingresos de un empleado, la aplicación asigna una referencia al objeto de empleado a una variable de la superclase Empleado, y después invoca al método ingresos en esa variable. Mantenemos un arreglo de variables Empleado, cada una de las cuales guarda una referencia a un objeto Empleado (desde luego que no puede haber objetos Empleado, ya que ésta es una clase abstracta; sin embargo, debido a la herencia todos los objetos de todas las subclases de Empleado pueden considerarse como objetos Empleado). El programa itera a través del arreglo y llama al método ingresos para cada objeto Empleado. Java procesa estas llamadas a los métodos en forma polimórfica. Al incluir a ingresos como un método abstracto en Empleado, se obliga a cada subclase directa de Empleado a sobrescribir el método ingresos para poder convertirse en una clase concreta. Esto permite al diseñador de la jerarquía de clases demandar que cada subclase concreta proporcione un cálculo apropiado del sueldo.

El método toString en la clase Empleado devuelve un objeto String que contiene el primer nombre, el apellido paterno y el número de seguro social del empleado. Como veremos, cada subclase de Empleado sobrescribe el método toString para crear una representación String de un objeto de esa clase que contiene el tipo del empleado (por ejemplo, "empleado asalariado:"), seguido del resto de la información del empleado.

El diagrama en la figura 10.3 muestra cada una de las cinco clases en la jerarquía, hacia abajo en la columna de la izquierda, y los métodos ingresos y toString en la fila superior. Para cada clase, el diagrama muestra los resultados deseados de cada método. [*Nota:* no listamos los métodos *establecer* y *obtener* de la superclase Empleado porque no se redefinen en ninguna de las subclases; cada una de estas propiedades se hereda y cada una de las subclases las utiliza "como están"].

Consideremos ahora la declaración de la clase Empleado (figura 10.4). Esta clase incluye un constructor que recibe el primer nombre, el apellido paterno y el número de seguro social como argumentos (líneas 11 a 16); los métodos *obtener* que devuelven el primer nombre, apellido y número de seguro social (líneas 25 a 28, 37 a 40 y 49 a 52, respectivamente); los métodos *establecer* que establecen el primer nombre, el apellido paterno y el número de seguro social (líneas 19 a 22, 31 a 34 y 43 a 46, respectivamente); el método toString (líneas 55 a 59), el cual devuelve la representación String de Empleado; y el método abstract ingresos (línea 62), que las subclases deben implementar. Observe que el constructor de Empleado no valida el número de seguro social en este ejemplo. Por lo general, se debe proporcionar esa validación.

¿Por qué declaramos a ingresos como un método abstracto? Simplemente, no tiene sentido proporcionar una implementación de este método en la clase Empleado. No podemos calcular los ingresos para un Empleado general; primero debemos conocer el tipo de Empleado específico para determinar el cálculo apropiado de los ingresos. Al declarar este método abstract, indicamos que cada subclase concreta *debe* proporcionar una implementación apropiada para ingresos, y que un programa podrá utilizar las variables de la superclase Empleado para invocar al método ingresos en forma polimórfica, para cualquier tipo de Empleado.

10.5.2 Creación de la subclase concreta EmpleadoAsalariado

La clase EmpleadoAsalariado (figura 10.5) extiende a la clase Empleado (línea 4) y redefine a ingresos (líneas 29 a 32), lo cual convierte a EmpleadoAsalariado en una clase concreta. La clase incluye un constructor (líneas 9 a 14) que recibe un primer nombre, un apellido paterno, un número de seguro social y un salario semanal como argumentos; un método *establecer* para asignar un valor positivo a la variable de instancia salarioSemanal (líneas 17 a 20); un método *obtener* para devolver el valor de salarioSemanal (líneas 23 a 26); un método ingresos (líneas 29 a 32) para calcular los ingresos de un EmpleadoAsalariado; y un método toString (líneas 35 a 39) que devuelve un objeto String que incluye el tipo del empleado, a saber, "empleado asalariado: ", seguido

	ingresos	toString
Empleado	abstract	*primerNombre apellidoPaterno* número de seguro social: *NSS*
Empleado-Asalariado	salarioSemanal	empleado asalariado: *primerNombre apellidoPaterno* número de seguro social: *NSS* salario semanal: *salarioSemanal*
EmpleadoPor-Horas	if horas <= 40 sueldo * horas else if horas > 40 40 * sueldo + (horas - 40) * sueldo * 1.5	empleado por horas: *primerNombre apellidoPaterno* número de seguro social: *NSS* sueldo por horas: *sueldo*; horas trabajadas: *horas*
EmpleadoPor-Comisión	tarifaComisión * ventasBrutas	empleado por comisión: *primerNombre apellidoPaterno* número de seguro social: *NSS* ventas brutas: *ventasBrutas*; tarifa de comisión: *tarifaComisión*
Empleado-BaseMas-Comision	(tarifaComision * ventasBrutas) + salarioBase	empleado por comisión con salario base: *primerNombre apellidoPaterno* número de seguro social: *NSS* ventas brutas: *ventasBrutas*; tarifa de comisión: *tarifaComision*; salario base: *salarioBase*

Figura 10.3 | Interfaz polimórfica para las clases de la jerarquía de `Empleado`.

```
// Fig. 10.4: Empleado.java
// La superclase abstracta Empleado.

public abstract class Empleado
{
   private String primerNombre;
   private String apellidoPaterno;
   private String numeroSeguroSocial;

   // constructor con tres argumentos
   public Empleado( String nombre, String apellido, String nss )
   {
      primerNombre = nombre;
      apellidoPaterno = apellido;
      numeroSeguroSocial = nss;
   } // fin del constructor de Empleado con tres argumentos

   // establece el primer nombre
   public void establecerPrimerNombre( String nombre )
   {
      primerNombre = nombre;
   } // fin del método establecerPrimerNombre

   // devuelve el primer nombre
   public String obtenerPrimerNombre()
```

Figura 10.4 | La superclase abstracta `Empleado`. (Parte 1 de 2).

```
   {
      return primerNombre;
   } // fin del método obtenerPrimerNombre

   // establece el apellido paterno
   public void establecerApellidoPaterno( String apellido )
   {
      apellidoPaterno = apellido;
   } // fin del método establecerApellidoPaterno

   // devuelve el apellido paterno
   public String obtenerApellidoPaterno()
   {
      return apellidoPaterno;
   } // fin del método obtenerApellidoPaterno

   // establece el número de seguro social
   public void establecerNumeroSeguroSocial( String nss )
   {
      numeroSeguroSocial = nss; // debe validar
   } // fin del método establecerNumeroSeguroSocial

   // devuelve el número de seguro social
   public String obtenerNumeroSeguroSocial()
   {
      return numeroSeguroSocial;
   } // fin del método obtenerNumeroSeguroSocial

   // devuelve representación String de un objeto Empleado
   public String toString()
   {
      return String.format( "%s %s\nnumero de seguro social: %s",
         obtenerPrimerNombre(), obtenerApellidoPaterno(), obtenerNumeroSeguroSocial() );
   } // fin del método toString

   // método abstracto sobrescrito por las subclases
   public abstract double ingresos(); // aquí no hay implementación
} // fin de la clase abstracta Empleado
```

Figura 10.4 | La superclase abstracta `Empleado`. (Parte 2 de 2).

```
// Fig. 10.5: EmpleadoAsalariado.java
// La clase EmpleadoAsalariado extiende a Empleado.

public class EmpleadoAsalariado extends Empleado
{
   private double salarioSemanal;

   // constructor de cuatro argumentos
   public EmpleadoAsalariado( String nombre, String apellido, String nss,
      double salario )
   {
      super( nombre, apellido, nss ); // los pasa al constructor de Empleado
      establecerSalarioSemanal( salario ); // valida y almacena el salario
   } // fin del constructor de EmpleadoAsalariado con cuatro argumentos

   // establece el salario
```

Figura 10.5 | La clase `EmpleadoAsalariado` derivada de `Empleado`. (Parte 1 de 2).

```
17     public void establecerSalarioSemanal( double salario )
18     {
19        salarioSemanal = salario < 0.0 ? 0.0 : salario;
20     } // fin del método establecerSalarioSemanal
21
22     // devuelve el salario
23     public double obtenerSalarioSemanal()
24     {
25        return salarioSemanal;
26     } // fin del método obtenerSalarioSemanal
27
28     // calcula los ingresos; sobrescribe el método abstracto ingresos en Empleado
29     public double ingresos()
30     {
31        return obtenerSalarioSemanal();
32     } // fin del método ingresos
33
34     // devuelve representación String de un objeto EmpleadoAsalariado
35     public String toString()
36     {
37        return String.format( "empleado asalariado: %s\n%s: $%,.2f",
38           super.toString(), "salario semanal", obtenerSalarioSemanal() );
39     } // fin del método toString
40  } // fin de la clase EmpleadoAsalariado
```

Figura 10.5 | La clase `EmpleadoAsalariado` derivada de `Empleado`. (Parte 2 de 2).

de la información específica para el empleado producida por el método `toString` de la superclase `Empleado` y el método `obtenerSalarioSemanal` de `EmpleadoAsalariado`. El constructor de la clase `EmpleadoAsalariado` pasa el primer nombre, el apellido paterno y el número de seguro social al constructor de `Empleado` (línea 12) para inicializar las variables de instancia `private` que no se heredan de la superclase. El método `ingresos` sobrescribe el método abstracto `ingresos` de `Empleado` para proporcionar una implementación concreta que devuelva el salario semanal del `EmpleadoAsalariado`. Si no implementamos `ingresos`, la clase `EmpleadoAsalariado` debe declararse como `abstract`; en caso contrario, se produce un error de compilación (y desde luego, queremos que `EmpleadoAsalariado` sea una clase concreta).

El método `toString` (líneas 35 a 39) de la clase `EmpleadoAsalariado` sobrescribe al método `toString` de `Empleado`. Si la clase `EmpleadoAsalariado` no sobrescribiera a `toString`, `EmpleadoAsalariado` habría heredado la versión de `toString` de `Empleado`. En ese caso, el método `toString` de `EmpleadoAsalariado` simplemente devolvería el nombre completo del empleado y su número de seguro social, lo cual no representa en forma adecuada a un `EmpleadoAsalariado`. Para producir una representación `String` completa de `EmpleadoAsalariado`, el método `toString` de la subclase devuelve `"empleado asalariado: "`, seguido de la información específica de la clase base `Empleado` (es decir, el primer nombre, el apellido paterno y el número de seguro social) que se obtiene al invocar el método `toString` de la superclase (línea 38); éste es un excelente ejemplo de reutilización de código. La representación `String` de un `EmpleadoAsalariado` también contiene el salario semanal del empleado, el cual se obtiene mediante la invocación del método `obtenerSalarioSemanal` de la clase.

10.5.3 Creación de la subclase concreta `EmpleadoPorHoras`

La clase `EmpleadoPorHoras` (figura 10.6) también extiende a `Empleado` (línea 4). La clase incluye un constructor (líneas 10 a 16) que recibe como argumentos un primer nombre, un apellido paterno, un número de seguro social, un sueldo por horas y el número de horas trabajadas. Las líneas 19 a 22 y 31 a 35 declaran los métodos *establecer* que asignan nuevos valores a las variables de instancia `sueldo` y `horas`, respectivamente. El método `establecerSueldo` (líneas 19 a 22) asegura que `sueldo` sea positivo, y el método `establecerHoras` (líneas 31 a 35) asegura que `horas` esté entre 0 y 168 (el número total de horas en una semana), inclusive. La clase `EmpleadoPorHoras` también incluye métodos *obtener* (líneas 25 a 28 y 38 a 41) para devolver los valores de `sueldo` y `horas`, respectivamente; un método `ingresos` (líneas 44 a 50) para calcular los ingresos de un `EmpleadoPorHo-`

```
// Fig. 10.6: EmpleadoPorHoras.java
// La clase EmpleadoPorHoras extiende a Empleado.

public class EmpleadoPorHoras extends Empleado
{
   private double sueldo; // sueldo por hora
   private double horas; // horas trabajadas por semana

   // constructor con cinco argumentos
   public EmpleadoPorHoras( String nombre, String apellido, String nss,
      double sueldoPorHoras, double horasTrabajadas )
   {
      super( nombre, apellido, nss );
      establecerSueldo( sueldoPorHoras ); // valida y almacena el sueldo por horas
      establecerHoras( horasTrabajadas ); // valida y almacena las horas trabajadas
   } // fin del constructor de EmpleadoPorHoras con cinco argumentos

   // establece el sueldo
   public void establecerSueldo( double sueldoPorHoras )
   {
      sueldo = ( sueldoPorHoras < 0.0 ) ? 0.0 : sueldoPorHoras;
   } // fin del método establecerSueldo

   // devuelve el sueldo
   public double obtenerSueldo()
   {
      return sueldo;
   } // fin del método obtenerSueldo

   // establece las horas trabajadas
   public void establecerHoras( double horasTrabajadas )
   {
      horas = ( ( horasTrabajadas >= 0.0 ) && ( horasTrabajadas <= 168.0 ) ) ?
         horasTrabajadas : 0.0;
   } // fin del método establecerHoras

   // devuelve las horas trabajadas
   public double obtenerHoras()
   {
      return horas;
   } // fin del método obtenerHoras

   // calcula los ingresos; sobrescribe el método abstracto ingresos en Empleado
   public double ingresos()
   {
      if ( obtenerHoras() <= 40 ) // no hay tiempo extra
         return obtenerSueldo() * obtenerHoras();
      else
         return 40 * obtenerSueldo() + ( obtenerHoras() - 40 ) * obtenerSueldo() * 1.5;
   } // fin del método ingresos

   // devuelve representación String de un objeto EmpleadoPorHoras
   public String toString()
   {
      return String.format( "empleado por horas: %s\n%s: $%,.2f; %s: %,.2f",
         super.toString(), "sueldo por hora", obtenerSueldo(),
         "horas trabajadas", obtenerHoras() );
   } // fin del método toString
} // fin de la clase EmpleadoPorHoras
```

Figura 10.6 | La clase `EmpleadoPorHoras` derivada de `Empleado`.

`ras`; y un método `toString` (líneas 53 a 58), que devuelve el tipo del empleado, a saber, `"empleado por horas: "`, e información específica para ese `Empleado`. Observe que el constructor de `EmpleadoPorHoras`, al igual que el constructor de `EmpleadoAsalariado`, pasa el primer nombre, el apellido paterno y el número de seguro social al constructor de la superclase `Empleado` (línea 13) para inicializar las variables de instancia `private`. Además, el método `toString` llama al método `toString` de la superclase (línea 56) para obtener la información específica del `Empleado` (es decir, primer nombre, apellido paterno y número de seguro social); éste es otro excelente ejemplo de reutilización de código.

10.5.4 Creación de la subclase concreta `EmpleadoPorComision`

La clase `EmpleadoPorComision` (figura 10.7) extiende a la clase `Empleado` (línea 4). Esta clase incluye a un constructor (líneas 10 a 16) que recibe como argumentos un primer nombre, un apellido, un número de seguro social, un monto de ventas y una tarifa de comisión; métodos *establecer* (líneas 19 a 22 y 31 a 34) para asignar nuevos valores a las variables de instancia `tarifaComision` y `ventasBrutas`, respectivamente; métodos *obtener* (líneas 25 a 28 y 37 a 40) que obtienen los valores de estas variables de instancia; el método `ingresos` (líneas 43 a 46) para calcular los ingresos de un `EmpleadoPorComision`; y el método `toString` (líneas 49 a 55) que devuelve el tipo del empleado, a saber, `"empleado por comisión: "`, e información específica del empleado. El constructor también pasa el primer nombre, el apellido y el número de seguro social al constructor de `Empleado` (línea 13) para inicializar las variables de instancia `private` de `Empleado`. El método `toString` llama al método `toString` de la superclase (línea 52) para obtener la información específica del `Empleado` (es decir, primer nombre, apellido paterno y número de seguro social).

```
1  // Fig. 10.7: EmpleadoPorComision.java
2  // La clase EmpleadoPorComision extiende a Empleado.
3
4  public class EmpleadoPorComision extends Empleado
5  {
6     private double ventasBrutas; // ventas totales por semana
7     private double tarifaComision; // porcentaje de comisión
8
9     // constructor con cinco argumentos
10    public EmpleadoPorComision( String nombre, String apellido, String nss,
11       double ventas, double tarifa )
12    {
13       super( nombre, apellido, nss );
14       establecerVentasBrutas( ventas );
15       establecerTarifaComision( tarifa );
16    } // fin del constructor de EmpleadoPorComision con cinco argumentos
17
18    // establece la tarifa de comisión
19    public void establecerTarifaComision( double tarifa )
20    {
21       tarifaComision = ( tarifa > 0.0 && tarifa < 1.0 ) ? tarifa : 0.0;
22    } // fin del método establecerTarifaComision
23
24    // devuelve la tarifa de comisión
25    public double obtenerTarifaComision()
26    {
27       return tarifaComision;
28    } // fin del método obtenerTarifaComision
29
30    // establece el monto de ventas brutas
31    public void establecerVentasBrutas( double ventas )
32    {
33       ventasBrutas = ( ventas < 0.0 ) ? 0.0 : ventas;
```

Figura 10.7 | La clase `EmpleadoPorComision` derivada de `Empleado`. (Parte 1 de 2).

```
   } // fin del método establecerVentasBrutas

   // devuelve el monto de ventas brutas
   public double obtenerVentasBrutas()
   {
      return ventasBrutas;
   } // fin del método obtenerVentasBrutas

   // calcula los ingresos; sobrescribe el método abstracto ingresos en Empleado
   public double ingresos()
   {
      return obtenerTarifaComision() * obtenerVentasBrutas();
   } // fin del método ingresos

   // devuelve representación String de un objeto EmpleadoPorComision
   public String toString()
   {
      return String.format( "%s: %s\n%s: $%,.2f; %s: %.2f",
         "empleado por comision", super.toString(),
         "ventas brutas", obtenerVentasBrutas(),
         "tarifa de comision", obtenerTarifaComision() );
   } // fin del método toString
} // fin de la clase EmpleadoPorComision
```

Figura 10.7 | La clase `EmpleadoPorComision` derivada de `Empleado`. (Parte 2 de 2).

10.5.5 Creación de la subclase concreta indirecta EmpleadoBaseMasComision

La clase `EmpleadoBaseMasComision` (figura 10.8) extiende a la clase `EmpleadoPorComision` (línea 4) y, por lo tanto, es una subclase indirecta de la clase `Empleado`. La clase `EmpleadoBaseMasComision` tiene un constructor (líneas 9 a 14) que recibe como argumentos un primer nombre, un apellido paterno, un número de seguro social, un monto de ventas, una tarifa de comisión y un salario base. Después pasa el primer nombre, el apellido paterno, el número de seguro social, el monto de ventas y la tarifa de comisión al constructor de `EmpleadoPorComision` (línea 12) para inicializar los miembros heredados. `EmpleadoBaseMasComision` también contiene un

```
1  // Fig. 10.8: EmpleadoBaseMasComision.java
2  // La clase EmpleadoBaseMasComision extiende a EmpleadoPorComision.
3
4  public class EmpleadoBaseMasComision extends EmpleadoPorComision
5  {
6     private double salarioBase; // salario base por semana
7
8     // constructor con seis argumentos
9     public EmpleadoBaseMasComision( String nombre, String apellido,
10       String nss, double ventas, double tarifa, double salario )
11    {
12       super( nombre, apellido, nss, ventas, tarifa );
13       establecerSalarioBase( salario ); // valida y almacena el salario base
14    } // fin del constructor de EmpleadoBaseMasComision con seis argumentos
15
16    // establece el salario base
17    public void establecerSalarioBase( double salario )
18    {
19       salarioBase = ( salario < 0.0 ) ? 0.0 : salario; // positivo
20    } // fin del método establecerSalarioBase
```

Figura 10.8 | La clase `EmpleadoBaseMasComision` derivada de `EmpleadoPorComision`. (Parte 1 de 2).

```
21
22     // devuelve el salario base
23     public double obtenerSalarioBase()
24     {
25        return salarioBase;
26     } // fin del método obtenerSalarioBase
27
28     // calcula los ingresos; sobrescribe el método ingresos en EmpleadoPorComision
29     public double ingresos()
30     {
31        return obtenerSalarioBase() + super.ingresos();
32     } // fin del método ingresos
33
34     // devuelve representación String de un objeto EmpleadoBaseMasComision
35     public String toString()
36     {
37        return String.format( "%s %s; %s: $%,.2f",
38           "con salario base", super.toString(),
39           "salario base", obtenerSalarioBase() );
40     } // fin del método toString
41  } // fin de la clase EmpleadoBaseMasComision
```

Figura 10.8 | La clase `EmpleadoBaseMasComision` derivada de `EmpleadoPorComision`. (Parte 2 de 2).

método *establecer* (líneas 17 a 20) para asignar un nuevo valor a la variable de instancia `salarioBase` y un método *obtener* (líneas 23 a 26) para devolver el valor de `salarioBase`. El método `ingresos` (líneas 29 a 32) calcula los ingresos de un `EmpleadoBaseMasComision`. Observe que la línea 31 en el método `ingresos` llama al método `ingresos` de la superclase `EmpleadoPorComision` para calcular la porción basada en la comisión de los ingresos del empleado. Éste es un buen ejemplo de reutilización de código. El método `toString` de `EmpleadoBaseMasComision` (líneas 35 a 40) crea una representación `String` de un `EmpleadoBaseMasComision`, la cual contiene `"con salario base"`, seguida del objeto `String` que se obtiene al invocar el método `toString` de la superclase `EmpleadoPorComision` (otro buen ejemplo de reutilización de código), y después el salario base. El resultado es un objeto `String` que empieza con `"con salario base empleado por comisión"`, seguido del resto de la información de `EmpleadoBaseMasComision`. Recuerde que el método `toString` de `EmpleadoPorComision` obtiene el primer nombre, el apellido paterno y el número de seguro social del empleado mediante la invocación del método `toString` de su superclase (es decir, `Empleado`); otro ejemplo más de reutilización de código. Observe que el método `toString` de `EmpleadoBaseMasComision` inicia una cadena de llamadas a métodos que abarcan los tres niveles de la jerarquía de `Empleado`.

10.5.6 Demostración del procesamiento polimórfico, el operador `instanceof` y la conversión descendente

Para probar nuestra jerarquía de `Empleado`, la aplicación en la figura 10.9 crea un objeto de cada una de las cuatro clases concretas `EmpleadoAsalariado`, `EmpleadoPorHoras`, `EmpleadoPorComision` y `EmpleadoBaseMasComision`. El programa manipula estos objetos, primero mediante variables del mismo tipo de cada objeto y después mediante el polimorfismo, utilizando un arreglo de variables `Empleado`. Al procesar los objetos mediante el polimorfismo, el programa incrementa el salario base de cada `EmpleadoBaseMasComision` en un 10% (desde luego que para esto se requiere determinar el tipo del objeto en tiempo de ejecución). Por último, el programa determina e imprime en forma polimórfica el tipo de cada objeto en el arreglo `Empleado`. Las líneas 9 a 18 crean objetos de cada una de las cuatro subclases concretas de `Empleado`. Las líneas 22 a 30 imprimen en pantalla la representación `String` y los ingresos de cada uno de estos objetos. Observe que `printf` llama en forma implícita al método `toString` de cada objeto, cuando éste se imprime en pantalla como un objeto `String` con el especificador de formato `%s`.

La línea 33 declara a `empleados` y le asigna un arreglo de cuatro variables `Empleado`. La línea 36 asigna la referencia a un objeto `EmpleadoAsalariado` a `empleados[ 0 ]`. La línea 37 asigna la referencia a un objeto

```
// Fig. 10.9: PruebaSistemaNomina.java
// Programa de prueba para la jerarquía de Empleado.

public class PruebaSistemaNomina
{
   public static void main( String args[] )
   {
      // crea objetos de las subclases
      EmpleadoAsalariado empleadoAsalariado =
         new EmpleadoAsalariado( "John", "Smith", "111-11-1111", 800.00 );
      EmpleadoPorHoras empleadoPorHoras =
         new EmpleadoPorHoras( "Karen", "Price", "222-22-2222", 16.75, 40 );
      EmpleadoPorComision empleadoPorComision =
         new EmpleadoPorComision(
         "Sue", "Jones", "333-33-3333", 10000, .06 );
      EmpleadoBaseMasComision empleadoBaseMasComision =
         new EmpleadoBaseMasComision(
         "Bob", "Lewis", "444-44-4444", 5000, .04, 300 );

      System.out.println( "Empleados procesados por separado:\n" );

      System.out.printf( "%s\n%s: $%,.2f\n\n",
         empleadoAsalariado, "ingresos", empleadoAsalariado.ingresos() );
      System.out.printf( "%s\n%s: $%,.2f\n\n",
         empleadoPorHoras, "ingresos", empleadoPorHoras.ingresos() );
      System.out.printf( "%s\n%s: $%,.2f\n\n",
         empleadoPorComision, "ingresos", empleadoPorComision.ingresos() );
      System.out.printf( "%s\n%s: $%,.2f\n\n",
         empleadoBaseMasComision,
         "ingresos", empleadoBaseMasComision.ingresos() );

      // crea un arreglo Empleado de cuatro elementos
      Empleado empleados[] = new Empleado[ 4 ];

      // inicializa el arreglo con objetos Empleado
      empleados[ 0 ] = empleadoAsalariado;
      empleados[ 1 ] = empleadoPorHoras;
      empleados[ 2 ] = empleadoPorComision;
      empleados[ 3 ] = empleadoBaseMasComision;

      System.out.println( "Empleados procesados en forma polimorfica:\n" );

      // procesa en forma genérica a cada elemento en el arreglo de empleados
      for ( Empleado empleadoActual : empleados )
      {
         System.out.println( empleadoActual ); // invoca a toString

         // determina si el elemento es un EmpleadoBaseMasComision
         if ( empleadoActual instanceof EmpleadoBaseMasComision )
         {
            // conversión descendente de la referencia de Empleado
            // a una referencia de EmpleadoBaseMasComision
            EmpleadoBaseMasComision empleado =
               ( EmpleadoBaseMasComision ) empleadoActual;

            double salarioBaseAnterior = empleado.obtenerSalarioBase();
            empleado.establecerSalarioBase( 1.10 * salarioBaseAnterior );
            System.out.printf(
               “el nuevo salario base con 10%% de aumento es : $%,.2f\n”,
```

Figura 10.9 | Programa de prueba de la jerarquía de clases de `Empleado`. (Parte 1 de 3).

```
                  empleado.obtenerSalarioBase() );
            } // fin de if

            System.out.printf(
               "ingresos $%,.2f\n\n", empleadoActual.ingresos() );
         } // fin de for

         // obtiene el nombre del tipo de cada objeto en el arreglo de empleados
         for ( int j = 0; j < empleados.length; j++ )
            System.out.printf( "El empleado %d es un %s\n", j,
               empleados[ j ].getClass().getName() );
      } // fin de main
   } // fin de la clase PruebaSistemaNomina
```

```
Empleados procesados por separado:

empleado asalariado: John Smith
numero de seguro social: 111-11-1111
salario semanal: $800.00
ingresos: $800.00

empleado por horas: Karen Price
numero de seguro social: 222-22-2222
sueldo por hora: $16.75; horas trabajadas: 40.00
ingresos: $670.00

empleado por comision: Sue Jones
numero de seguro social: 333-33-3333
ventas brutas: $10,000.00; tarifa de comision: 0.06
ingresos: $600.00

con salario base empleado por comision: Bob Lewis
numero de seguro social: 444-44-4444
ventas brutas: $5,000.00; tarifa de comision: 0.04; salario base: $300.00
ingresos: $500.00

Empleados procesados en forma polimorfica:

empleado asalariado: John Smith
numero de seguro social: 111-11-1111
salario semanal: $800.00
ingresos $800.00

empleado por horas: Karen Price
numero de seguro social: 222-22-2222
sueldo por hora: $16.75; horas trabajadas: 40.00
ingresos $670.00

empleado por comision: Sue Jones
numero de seguro social: 333-33-3333
ventas brutas: $10,000.00; tarifa de comision: 0.06
ingresos $600.00

con salario base empleado por comision: Bob Lewis
numero de seguro social: 444-44-4444
ventas brutas: $5,000.00; tarifa de comision: 0.04; salario base: $300.00
el nuevo salario base con 10% de aumento es : $330.00
ingresos $530.00
```

Figura 10.9 | Programa de prueba de la jerarquía de clases de `Empleado`. (Parte 2 de 3).

```
El empleado 0 es un EmpleadoAsalariado
El empleado 1 es un EmpleadoPorHoras
El empleado 2 es un EmpleadoPorComision
El empleado 3 es un EmpleadoBaseMasComision
```

Figura 10.9 | Programa de prueba de la jerarquía de clases de `Empleado`. (Parte 3 de 3).

`EmpleadoPorHoras` a `empleados[ 1 ]`. La línea 38 asigna la referencia a un objeto `EmpleadoPorComision` a `empleados[ 2 ]`. La línea 39 asigna la referencia a un objeto `EmpleadoBaseMasComision` a `empleados[ 3 ]`. Cada asignación es permitida, ya que un `EmpleadoAsalariado` *es un* `Empleado`, un `EmpleadoPorHoras` *es un* `Empleado`, un `EmpleadoPorComision` *es un* `Empleado` y un `EmpleadoBaseMasComision` *es un* `Empleado`. Por lo tanto, podemos asignar las referencias de los objetos `EmpleadoAsalariado`, `EmpleadoPorHoras`, `EmpleadoPorComision` y `EmpleadoBaseMasComision` a variables de la superclase `Empleado`, aun cuando ésta es una clase abstracta.

Las líneas 44 a 65 iteran a través del arreglo `empleados` e invocan los métodos `toString` e `ingresos` con la variable `empleadoActual` de `Empleado`, a la cual se le asigna la referencia a un `Empleado` distinto en el arreglo, durante cada iteración. Los resultados ilustran que en definitivo se invocan los métodos apropiados para cada clase. Todas las llamadas a los métodos `toString` e `ingresos` se resuelven en tiempo de ejecución, con base en el tipo del objeto al que `empleadoActual` hace referencia. Este proceso se conoce como **vinculación dinámica** o **vinculación postergada**. Por ejemplo, la línea 46 invoca en forma implícita al método `toString` del objeto al que `empleadoActual` hace referencia. Como resultado de la vinculación dinámica, Java decide qué método `toString` de cuál clase llamará en tiempo de ejecución, en vez de hacerlo en tiempo de compilación. Observe que sólo los métodos de la clase `Empleado` pueden llamarse a través de una variable `Empleado` (y desde luego que `Empleado` incluye los métodos de la clase `Object`). (En la sección 9.7 vimos el conjunto de métodos que todas las clases heredan de la clase `Object`). Una referencia a la superclase puede utilizarse para invocar sólo a métodos de la superclase (y la superclase puede invocar versiones sobrescritas de éstos en la subclase).

Realizamos un procesamiento especial en los objetos `EmpleadoBasePorComision`; a medida que los encontramos, incrementamos su salario base en un 10%. Cuando procesamos objetos en forma polimórfica, por lo general no necesitamos preocuparnos por los "detalles específicos", pero para ajustar el salario base, tenemos que determinar el tipo específico de cada objeto `Empleado` en tiempo de ejecución. La línea 49 utiliza el operador **`instanceof`** para determinar si el tipo de cierto objeto `Empleado` es `EmpleadoBaseMasComision`. La condición en la línea 49 es verdadera si el objeto al que hace referencia `empleadoActual` *es un* `EmpleadoBaseMasComision`. Esto también sería verdadero para cualquier objeto de una subclase de `EmpleadoBaseMasComision`, debido a la relación *"es un"* que tiene una subclase con su superclase. Las líneas 53 y 54 realizan una conversión descendente en `empleadoActual`, del tipo `Empleado` al tipo `EmpleadoBaseMasComision`; esta conversión se permite sólo si el objeto tiene una relación *"es un"* con `EmpleadoBaseMasComision`. La condición en la línea 49 asegura que éste sea el caso. Esta conversión se requiere si vamos a invocar los métodos `obtenerSalarioBase` y `establecerSalarioBase` de la subclase `EmpleadoBaseMasComision` en el objeto `Empleado` actual; como veremos en un momento, si tratamos de invocar a un método que pertenezca sólo a la subclase directamente en una referencia a la superclase, se produce un error de compilación.



Error común de programación 10.3

Asignar una variable de la superclase a una variable de la subclase (sin una conversión descendente explícita) es un error de compilación.



Observación de ingeniería de software 10.5

Si en tiempo de ejecución se asigna la referencia a un objeto de la subclase a una variable de una de sus superclases directas o indirectas, es aceptable convertir la referencia almacenada en esa variable de la superclase, de vuelta a una referencia del tipo de la subclase. Antes de realizar dicha conversión, use el operador `instanceof` para asegurar que el objeto sea indudablemente de un tipo de subclase apropiado.



Error común de programación 10.4

Al realizar una conversión descendente sobre un objeto, se produce una excepción `ClassCastException` *si, en tiempo de ejecución, el objeto no tiene una relación* "es un" *con el tipo especificado en el operador de conversión. Un objeto puede convertirse sólo a su propio tipo, o al tipo de una de sus superclases.*

Si la expresión `instanceof` en la línea 49 es `true`, el cuerpo de la instrucción `if` (líneas 49 a 61) realiza el procesamiento especial requerido para el objeto `EmpleadoBaseMasComision`. Usando la variable `empleado` de `EmpleadoBaseMasComision`, las líneas 56 y 57 invocan a los métodos `obtenerSalarioBase` y `establecerSalarioBase`, que sólo pertenecen a la subclase, para obtener y actualizar el salario base del empleado con el aumento del 10%.

Las líneas 63 y 64 invocan al método `ingresos` en `empleadoActual`, el cual llama al método `ingresos` del objeto de la subclase apropiada en forma polimórfica. Como puede ver, al obtener en forma polimórfica los ingresos del `EmpleadoAsalariado`, el `EmpleadoPorHoras` y el `EmpleadoPorComision` en las líneas 63 y 64, se produce el mismo resultado que obtener los ingresos de estos empleados en forma individual, en las líneas 22 a 27. No obstante, el monto de los ingresos obtenidos para el `EmpleadoBaseMasComision` en las líneas 63 y 64 es más alto que el que se obtiene en las líneas 28 a 30, debido al aumento del 10% en su salario base.

Las líneas 68 a 70 imprimen en pantalla el tipo de cada empleado, como un objeto `String`. Todos los objetos en Java conocen su propia clase y pueden acceder a esta información a través del método **`getClass`**, que todas las clases heredan de la clase `Object`. El método `getClass` devuelve un objeto de tipo **`Class`** (del paquete `java.lang`), el cual contiene información acerca del tipo del objeto, incluyendo el nombre de su clase. La línea 70 invoca al método `getClass` en el objeto para obtener su clase en tiempo de ejecución (es decir, un objeto `Class` que representa el tipo del objeto). Después se invoca el método **`getName`** en el objeto devuelto por `getClass`, para obtener el nombre de la clase. Para aprender más acerca de la clase `Class`, consulte su documentación en línea, en `java.sun.com/javase/6/docs/api/java/lang/Class.html`.

En el ejemplo anterior, evitamos varios errores de compilación mediante la conversión descendente de una variable de `Empleado` a una variable de `EmpleadoBaseMasComision` en las líneas 53 y 54. Si eliminamos el operador de conversión `( EmpleadoBaseMasComision )` de la línea 54 y tratamos de asignar la variable `empleadoActual` de `Empleado` directamente a la variable `empleado` de `EmpleadoBaseMasComision`, recibiremos un error de compilación del tipo "`incompatible types`" (incompatibilidad de tipos). Este error indica que el intento de asignar la referencia del objeto `empleadoPorComision` de la superclase a la variable `empleadoBaseMasComision` de la subclase no se permite. El compilador evita esta asignación debido a que un `EmpleadoPorComision` no es un `EmpleadoBaseMasComision`; la relación *"es un"* se aplica sólo entre la subclase y sus superclases, no viceversa.

De manera similar, si las líneas 56, 57 y 60 utilizaran la variable `empleadoActual` de la superclase en vez de la variable `empleado` de la subclase, para invocar a los métodos `obtenerSalarioBase` y `establecerSalarioBase` que sólo pertenecen a la subclase, recibiríamos un error de compilación del tipo "`cannot find symbol`" (no se puede encontrar el símbolo) en cada una de estas líneas. No se permite tratar de invocar métodos que pertenezcan sólo a la subclase en una referencia a la superclase. Mientras que las líneas 56, 57 y 60 se ejecutan sólo si `instanceof` en la línea 49 devuelve `true` para indicar que a `empleadoActual` se le asignó una referencia a un objeto `EmpleadoBaseMasComision`, no podemos tratar de invocar los métodos `obtenerSalarioBase` y `establecerSalarioBase` de la subclase `EmpleadoBaseMasComision` en la referencia `empleadoActual` de la superclase `Empleado`. El compilador generaría errores en las líneas 56, 57 y 60, ya que `obtenerSalarioBase` y `establecerSalarioBase` no son métodos de la superclase y no pueden invocarse en una variable de la superclase. Aunque el método que se vaya a llamar en realidad depende del tipo del objeto en tiempo de ejecución, puede utilizarse una variable para invocar sólo a los métodos que sean miembros del tipo de esa variable, lo cual verifica el compilador. Si utilizamos una variable `Empleado` de la superclase, sólo podemos invocar a los métodos que se encuentran en la clase `Empleado: ingresos`, `toString`, y los métodos *obtener* y *establecer* de `Empleado`.

10.5.7 Resumen de las asignaciones permitidas entre variables de la superclase y de la subclase

Ahora que hemos visto una aplicación completa que procesa diversos objetos de las subclases en forma polimórfica, sintetizaremos lo que puede y no puede hacer con los objetos y variables de las superclases y las subclases. Aunque un objeto de una subclase también *es un* objeto de su superclase, los dos objetos son, sin embargo, distintos. Como vimos antes, los objetos de una subclase pueden tratarse como si fueran objetos de la superclase. Pero

como la subclase puede tener miembros adicionales que sólo pertenezcan a esa subclase, no se permite asignar una referencia de la superclase a una variable de la subclase sin una conversión explícita; dicha asignación dejaría los miembros de la subclase indefinidos para el objeto de la superclase.

En esta sección y en la sección 10.3, además del capítulo 9, hemos visto cuatro maneras de asignar referencias de una superclase y de una subclase a las variables de los tipos de la superclase y la subclase:

1. Asignar una referencia de la superclase a una variable de la superclase es un proceso simple y directo.
2. Asignar una referencia de la subclase a una variable de la subclase es un proceso simple y directo.
3. Asignar una referencia de la subclase a una variable de la superclase es seguro, ya que el objeto de la subclase *es un* objeto de su superclase. No obstante, esta referencia puede usarse para referirse sólo a los miembros de la superclase. Si este código hace referencia a los miembros que pertenezcan sólo a la subclase, a través de la variable de la superclase, el compilador reporta errores.
4. Tratar de asignar una referencia de la superclase a una variable de la subclase produce un error de compilación. Para evitar este error, la referencia de la superclase debe convertirse en forma explícita a un tipo de la subclase. En tiempo de ejecución, si el objeto al que se refiere la referencia no es un objeto de la subclase, se producirá una excepción. (Para más información sobre el manejo de excepciones, vea el capítulo 13, Manejo de excepciones). El operador `instanceof` puede utilizarse para asegurar que dicha conversión se realice sólo si el objeto es de la subclase.

10.6 Métodos y clases final

En la sección 6.10 vimos que las variables pueden declararse como `final` para indicar que no pueden modificarse una vez que se inicializan; dichas variables representan valores constantes. También es posible declarar métodos, parámetros de los métodos y clases con el modificador `final`.

Un método que se declara como `final` en una superclase no puede sobrescribirse en una subclase. Los métodos que se declaran como `private` son implícitamente `final`, ya que es imposible sobrescribirlos en una subclase. Los métodos que se declaran como `static` son implícitamente `final`. La declaración de un método `final` nunca puede cambiar, por lo cual todas las subclases utilizan la misma implementación del método, y las llamadas a los métodos `final` se resuelven en tiempo de compilación; a esto se le conoce como **vinculación estática**. Como el compilador sabe que los métodos `final` no se pueden sobrescribir, puede optimizar los programas eliminando las llamadas a los métodos `final`, y reemplazándolas con el código expandido de sus declaraciones en la ubicación de cada una de las llamadas a los métodos; a esta técnica se le conoce como **poner el código en línea**.



Tip de rendimiento 10.1

El compilador puede decidir poner en línea la llamada a un método `final` y lo hará para los métodos `final` pequeños y simples. La puesta en línea no quebranta los principios del encapsulamiento o de ocultamiento de la información, pero sí mejora el rendimiento, ya que elimina la sobrecarga que se produce al realizar la llamada a un método.

Una clase que se declara como `final` no puede ser una superclase (es decir, una clase no puede extender a una clase `final`). Todos los métodos en una clase `final` son implícitamente `final`. La clase `String` es un ejemplo de una clase `final`. Esta clase no puede extenderse, por lo que los programas que utilizan objetos `String` pueden depender de la funcionalidad de los objetos `String`, según lo especificado en la API de java. Al hacer la clase `final` también se evita que los programadores creen subclases que podrían ignorar las restricciones de seguridad. Para obtener más información sobre las clases y métodos `final`, visite `java.sun.com/docs/books/tutorial/java/IandI/final.html`. Este sitio contiene información adicional acerca de cómo usar clases `final` para mejorar la seguridad de un sistema.



Error común de programación 10.5

Tratar de declarar una subclase de una clase `final` es un error de compilación.



Observación de ingeniería de software 10.6

En la API de Java, la vasta mayoría de clases no se declara como `final`. Esto permite la herencia y el polimorfismo: las características fundamentales de la programación orientada a objetos. Sin embargo, en algunos casos es importante declarar las clases como `final`; generalmente por razones de seguridad.

10.7 Ejemplo práctico: creación y uso de interfaces

En nuestro siguiente ejemplo (figuras 10.11 a 10.13) analizaremos nuevamente el sistema de nómina de la sección 10.5. Suponga que la compañía involucrada desea realizar varias operaciones de contabilidad en una sola aplicación de cuentas por pagar; además de calcular los ingresos de nómina que deben pagarse a cada empleado, la compañía debe también calcular el pago vencido en cada una de varias facturas (por los bienes comprados). Aunque se aplican a cosas no relacionadas (es decir, empleados y facturas), ambas operaciones tienen que ver con el cálculo de algún tipo de monto a pagar. Para un empleado, el pago se refiere a sus ingresos. Para una factura, el pago se refiere al costo total de los bienes listados en la misma. ¿Podemos calcular esas cosas distintas, como los pagos vencidos para los empleados y las facturas, en forma polimórfica en una sola aplicación? ¿Ofrece Java una herramienta que requiera que las clases no relacionadas implementen un conjunto de métodos comunes (por ejemplo, un método que calcule un monto a pagar)? Las **interfaces** de Java ofrecen exactamente esta herramienta.

Las interfaces definen y estandarizan las formas en que pueden interactuar las cosas entre sí, como las personas y los sistemas. Por ejemplo, los controles en un radio sirven como una interfaz entre los usuarios del radio y sus componentes internos. Los controles permiten a los usuarios realizar un conjunto limitado de operaciones (por ejemplo, cambiar la estación, ajustar el volumen, seleccionar AM o FM), y distintos radios pueden implementar los controles de distintas formas (por ejemplo, el uso de botones, perillas, comandos de voz). La interfaz especifica *qué* operaciones debe permitir el radio que realicen los usuarios, pero no especifica *cómo* deben realizarse las operaciones. De manera similar, la interfaz entre un conductor y un automóvil con transmisión manual incluye el volante, la palanca de cambios, el pedal del embrague, el pedal del acelerador y el pedal del freno. Esta misma interfaz se encuentra en casi todos los automóviles de transmisión manual, lo que permite que alguien que sabe cómo manejar cierto automóvil de transmisión manual sepa cómo manejar casi cualquier automóvil de transmisión manual. Los componentes de cada automóvil individual pueden tener una apariencia ligeramente distinta, pero el propósito general es el mismo; permitir que las personas conduzcan el automóvil.

Los objetos de software también se comunican a través de interfaces. Una interfaz de Java describe un conjunto de métodos que pueden llamarse sobre un objeto, para indicar al objeto que realice cierta tarea, o que devuelva cierta pieza de información, por ejemplo. El siguiente ejemplo introduce una interfaz llamada `PorPagar`, la cual describe la funcionalidad de cualquier objeto que deba ser capaz de recibir un pago y, por lo tanto, debe ofrecer un método para determinar el monto de pago vencido apropiado. La **declaración de una interfaz** empieza con la palabra clave **`interface`** y sólo puede contener constantes y métodos `abstract`. A diferencia de las clases, todos los miembros de la interfaz deben ser `public`, y las interfaces no pueden especificar ningún detalle de implementación, como las declaraciones de métodos concretos y variables de instancia. Por lo tanto, todos los métodos que se declaran en una interfaz son `public abstract` de manera implícita, y todos los campos son implícitamente `public`, `static` y `final`.



Buena práctica de programación 10.1

De acuerdo con el capítulo 9 de la Especificación del lenguaje Java, es un estilo apropiado declarar los métodos de una interfaz sin las palabras clave `public` y `abstract`, ya que son redundantes en las declaraciones de los métodos de la interfaz. De manera similar, las constantes deben declararse sin las palabras clave `public`, `static` y `final`, ya que también son redundantes.

Para utilizar una interfaz, una clase debe especificar que implementa (**`implements`**) a esa interfaz y debe declarar cada uno de sus métodos con la firma especificada en la declaración de la interfaz. Una clase que no implementa a todos los métodos de la interfaz es una clase abstracta, y debe declararse como `abstract`. Implementar una interfaz es como firmar un contrato con el compilador que diga, "Declararé todos los métodos especificados por la interfaz, o declararé mi clase como `abstract`".



Error común de programación 10.6

Si no declaramos ningún miembro de una interfaz en una clase concreta que implemente (`implements`) a esa interfaz, se produce un error de compilación indicando que la clase debe declararse como `abstract`.

Por lo general, una interfaz se utiliza cuando clases dispares (es decir, no relacionadas) necesitan compartir métodos y constantes comunes. Esto permite que los objetos de clases no relacionadas se procesen en forma polimórfica; los objetos de clases que implementan la misma interfaz pueden responder a las mismas llamadas a métodos. Usted puede crear una interfaz que describa la funcionalidad deseada y después implementar esta

interfaz en cualquier clase que requiera esa funcionalidad. Por ejemplo, en la aplicación de cuentas por pagar que desarrollaremos en esta sección, implementamos la interfaz `PorPagar` en cualquier clase que deba tener la capacidad de calcular el monto de un pago (por ejemplo, `Empleado`, `Factura`).

A menudo, una interfaz se utiliza en vez de una clase `abstract` cuando no hay una implementación predeterminada que heredar; esto es, no hay campos ni implementaciones de métodos predeterminadas. Al igual que las clases `public abstract`, las interfaces son comúnmente de tipo `public`, por lo que se declaran en archivos por sí solas con el mismo nombre que la interfaz, y la extensión de archivo `.java`.

10.7.1 Desarrollo de una jerarquía PorPagar

Para crear una aplicación que pueda determinar los pagos para los empleados y facturas por igual, primero crearemos una interfaz llamada `Porpagar`, la cual contiene el método `obtenerMontoPago`, que devuelve un monto `double` que debe pagarse para un objeto de cualquier clase que implemente a la interfaz. El método `obtenerMontoPago` es una versión de propósito general del método `ingresos` de la jerarquía de `Empleado`; el método `ingresos` calcula un monto de pago específicamente para un `Empleado`, mientras que `obtenerMontoPago` puede aplicarse a un amplio rango de objetos no relacionados. Después de declarar la interfaz `PorPagar` presentaremos la clase `Factura`, la cual implementa a la interfaz `PorPagar`. Luego modificaremos la clase `Empleado` de tal forma que también implemente a la interfaz `PorPagar`. Por último, actualizaremos la subclase `EmpleadoAsalariado` de `Empleado` para "ajustarla" en la jerarquía de `PorPagar` (es decir, cambiaremos el nombre del método `ingresos` de `EmpleadoAsalariado` por el de `obtenerMontoPago`).



Buena práctica de programación 10.2

Al declarar un método en una interfaz, seleccione un nombre para el método que describa su propósito en forma general, ya que el método podría implementarse por muchas clases no relacionadas.

Las clases `Factura` y `Empleado` representan cosas para las cuales la compañía debe calcular un monto a pagar. Ambas clases implementan a `PorPagar`, por lo que un programa puede invocar al método `obtenerMontoPago` en objetos `Factura` y `Empleado` por igual. Como pronto veremos, esto permite el procesamiento polimórfico de objetos `Factura` y `Empleado` requerido para la aplicación de cuentas por pagar de nuestra compañía.

El diagrama de clases de UML en la figura 10.10 muestra la jerarquía utilizada en nuestra aplicación de cuentas por pagar. La jerarquía comienza con la interfaz `PorPagar`. UML diferencia a una interfaz de otras clases colocando la palabra "interface" entre los signos « y », por encima del nombre de la interfaz. UML expresa la relación entre una clase y una interfaz a través de una relación conocida como **realización**. Se dice que una clase "realiza", o implementa, los métodos de una interfaz. Un diagrama de clases modela una realización como una flecha punteada que parte de la clase que realizará la implementación, hasta la interfaz. El diagrama en la figura 10.10 indica que cada una de las clases `Factura` y `Empleado` pueden realizar (es decir, implementar) la interfaz `PorPagar`. Observe que, al igual que en el diagrama de clases de la figura 10.2, la clase `Empleado` aparece en cursivas, lo cual indica que es una clase abstracta. La clase concreta `EmpleadoAsalariado` extiende a `Empleado` y hereda la relación de realización de su superclase con la interfaz `PorPagar`.

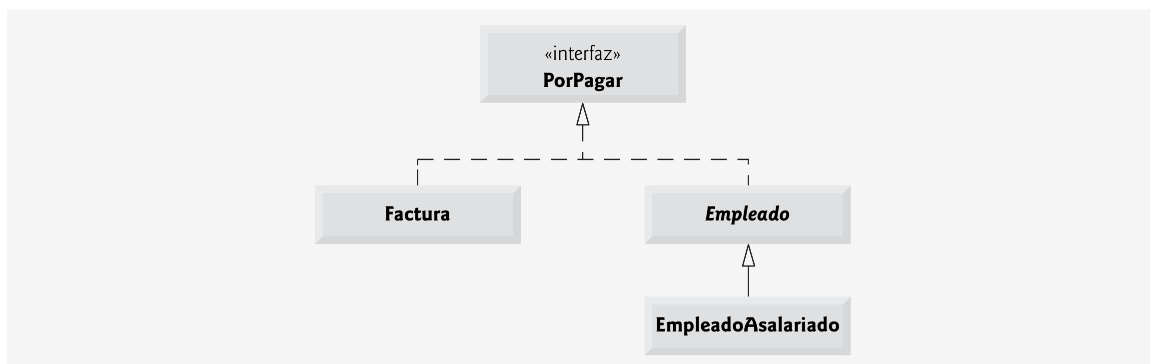


Figura 10.10 | Diagrama de clases de UML de la jerarquía de la interfaz `PorPagar`.

10.7.2 Declaración de la interfaz PorPagar

La declaración de la interfaz `PorPagar` empieza en la figura 10.11, línea 4. La interfaz `PorPagar` contiene el método `public abstract obtenerMontoPago` (línea 6). Observe que este método no puede declararse en forma explícita como `public` o `abstract`. Los métodos de una interfaz deben ser `public` y `abstract`, por lo cual no necesitan declararse como tales. La interfaz `PorPagar` sólo tiene un método; las interfaces pueden tener cualquier número de métodos. (Más adelante en el libro veremos la noción de las "interfaces de marcado"; en realidad éstas *no* tienen métodos. De hecho, una interfaz de marcado no contiene valores constantes; simplemente contiene una declaración vacía). Además, el método `obtenerMontoPago` no tiene parámetros, pero los métodos de las interfaces pueden tener parámetros.

```
1  // Fig. 10.11: PorPagar.java
2  // Declaración de la interfaz PorPagar.
3
4  public interface PorPagar
5  {
6     double obtenerMontoPago(); // calcula el pago; no hay implementación
7  } // fin de la interfaz PorPagar
```

Figura 10.11 | Declaración de la interfaz `PorPagar`.

10.7.3 Creación de la clase Factura

Ahora crearemos la clase `Factura` (figura 10.12) para representar una factura simple que contiene información de facturación para cierto tipo de pieza. La clase declara las variables de instancia `private numeroPieza`, `descripcionPieza`, `cantidad` y `precioPorArticulo` (líneas 6 a 9), las cuales indican el número de pieza, su descripción, la cantidad de piezas ordenadas y el precio por artículo. La clase `Factura` también contiene un constructor (líneas 12 a 19), métodos *obtener* y *establecer* (líneas 22 a 67) que manipulan las variables de instancia de la clase y un método `toString` (líneas 70 a 75) que devuelve una representación `String` de un objeto `Factura`. Observe que los métodos `establecerCantidad` (líneas 46 a 49) y `establecerPrecioPorArticulo` (líneas 58 a 61) aseguran que `cantidad` y `precioPorArticulo` obtengan sólo valores positivos.

La línea 4 de la figura 10.12 indica que la clase `Factura` implementa a la interfaz `PorPagar`. Al igual que todas las clases, la clase `Factura` también extiende a `Object` de manera implícita. Java no permite que las subclases hereden de más de una superclase, pero sí permite que una clase herede de una superclase e implemente más de una interfaz. De hecho, una clase puede implementar todas las interfaces que necesite, además de extender otra clase. Para implementar más de una interfaz, utilice una lista separada por comas de nombres de interfaz después de la palabra clave `implements` en la declaración de la clase, como se muestra a continuación:

`public class` *NombreClase* `extends` *NombreSuperClase* `implements` *PrimeraInterfaz*`,` *SegundaInterfaz*, …

Todos los objetos de una clase que implementan varias interfaces tienen la relación *"es un"* con cada tipo de interfaz implementada.

```
1  // Fig. 10.12: Factura.java
2  // La clase Factura implementa a PorPagar.
3
4  public class Factura implements PorPagar
5  {
6     private String numeroPieza;
7     private String descripcionPieza;
8     private int cantidad;
9     private double precioPorArticulo;
```

Figura 10.12 | La clase `Factura`, que implementa a `Porpagar`. (Parte 1 de 3).

```java

   // constructor con cuatro argumentos
   public Factura( String pieza, String descripcion, int cuenta,
      double precio )
   {
      numeroPieza = pieza;
      descripcionPieza = descripcion;
      establecerCantidad( cuenta ); // valida y almacena la cantidad
      establecerPrecioPorArticulo( precio ); // valida y almacena el precio por artículo
   } // fin del constructor de Factura con cuatro argumentos

   // establece el número de pieza
    public void establecerNumeroPieza( String pieza )
   {
      numeroPieza = pieza;
   } // fin del método establecerNumeroPieza

   // obtener número de pieza
   public String obtenerNumeroPieza()
   {
      return numeroPieza;
   } // fin del método obtenerNumeroPieza

   // establece la descripción
   public void establecerDescripcionPieza( String descripcion )
   {
      descripcionPieza = descripcion;
   } // fin del método establecerDescripcionPieza

   // obtiene la descripción
   public String obtenerDescripcionPieza()
   {
      return descripcionPieza;
   } // fin del método obtenerDescripcionPieza

   // establece la cantidad
   public void establecerCantidad( int cuenta )
   {
      cantidad = ( cuenta < 0 ) ? 0 : cuenta; // cantidad no puede ser negativa
   } // fin del método establecerCantidad

   // obtener cantidad
   public int obtenerCantidad()
   {
      return cantidad;
   } // fin del método obtenerCantidad

   // establece el precio por artículo
   public void establecerPrecioPorArticulo( double precio )
   {
      precioPorArticulo = ( precio < 0.0 ) ? 0.0 : precio; // valida el precio
   } // fin del método establecerPrecioPorArticulo

   // obtiene el precio por artículo
   public double obtenerPrecioPorArticulo()
   {
      return precioPorArticulo;
   } // fin del método obtenerPrecioPorArticulo

```

Figura 10.12 | La clase `Factura`, que implementa a `Porpagar`. (Parte 2 de 3).

```
69     // devuelve representación String de un objeto Factura
70     public String toString()
71     {
72        return String.format( "%s: \n%s: %s (%s) \n%s: %d \n%s: $%,.2f",
73           "factura", "numero de pieza", obtenerNumeroPieza(), obtenerDescripcionPieza(),
74           "cantidad", obtenerCantidad(), "precio por articulo", obtenerPrecioPorArticulo() );
75     } // fin del método toString
76
77     // método requerido para realizar el contrato con la interfaz PorPagar
78     public double obtenerMontoPago()
79     {
80        return obtenerCantidad() * obtenerPrecioPorArticulo(); // calcula el costo total
81     } // fin del método obtenerMontoPago
82  } // fin de la clase Factura
```

Figura 10.12 | La clase `Factura`, que implementa a `Porpagar`. (Parte 3 de 3).

La clase `Factura` implementa el único método de la interfaz `PorPagar`. El método `obtenerMontoPago` se declara en las líneas 78 a 81. Este método calcula el pago total requerido para pagar la factura. El método multiplica los valores de `cantidad` y `precioPorArticulo` (que se obtienen a través de los métodos *obtener* apropiados) y devuelve el resultado (línea 80). Este método cumple con el requerimiento de implementación del mismo en la interfaz `PorPagar`; hemos cumplido el contrato de interfaz con el compilador.

10.7.4 Modificación de la clase `Empleado` para implementar la interfaz `PorPagar`

Ahora modificaremos la clase `Empleado` para que implemente la interfaz `PorPagar`. La figura 10.13 contiene la clase `Empleado` modificada. Esta declaración de la clase es idéntica a la de la figura 10.4, con sólo dos excepciones. En primer lugar, la línea 4 de la figura 10.13 indica que la clase `Empleado` ahora implementa a la interfaz `PorPagar`. En segundo lugar, como `Empleado` ahora implementa a la interfaz `PorPagar`, debemos cambiar el nombre de `ingresos` por el de `obtenerMontoPago` en toda la jerarquía de `Empleado`. Sin embargo, al igual que con el método `ingresos` en la versión de la clase `Empleado` de la figura 10.4, no tiene sentido implementar el método `obtenerMontoPago` en la clase `Empleado`, ya que no podemos calcular el pago de los ingresos para un `Empleado` general; primero debemos conocer el tipo específico de `Empleado`. En la figura 10.4 declaramos el método `ingresos` como `abstract` por esta razón y, como resultado, la clase `Empleado` tuvo que declararse como `abstract`. Esto obliga a cada clase derivada de `Empleado` a redefinir `ingresos` con una implementación concreta.

```
1  // Fig. 10.13: Empleado.java
2  // La superclases abstracta Empleado implementa a PorPagar.
3
4  public abstract class Empleado implements PorPagar
5  {
6     private String primerNombre;
7     private String apellidoPaterno;
8     private String numeroSeguroSocial;
9
10    // constructor con tres argumentos
11    public Empleado( String nombre, String apellido, String nss )
12    {
13       primerNombre = nombre;
14       apellidoPaterno = apellido;
15       numeroSeguroSocial = nss;
16    } // fin del constructor de Empleado con tres argumentos
```

Figura 10.13 | La clase `Empleado`, que implementa a `PorPagar`. (Parte 1 de 2).

```
17
18     // establece el primer nombre
19     public void establecerPrimerNombre( String nombre )
20     {
21        primerNombre = nombre;
22     } // fin del método establecerPrimerNombre
23
24     // devuelve el primer nombre
25     public String obtenerPrimerNombre()
26     {
27        return primerNombre;
28     } // fin del método obtenerPrimerNombre
29
30     // establece el apellido paterno
31     public void establecerApellidoPaterno( String apellido )
32     {
33        apellidoPaterno = apellido;
34     } // fin del método establecerApellidoPaterno
35
36     // devuelve el apellido paterno
37     public String obtenerApellidoPaterno()
38     {
39        return apellidoPaterno;
40     } // fin del método obtenerApellidoPaterno
41
42     // establece el número de seguro social
43     public void establecerNumeroSeguroSocial( String nss )
44     {
45        numeroSeguroSocial = nss; // debe validar
46     } // fin del método establecerNumeroSeguroSocial
47
48     // devuelve el número de seguro social
49     public String obtenerNumeroSeguroSocial()
50     {
51        return numeroSeguroSocial;
52     } // fin del método obtenerNumeroSeguroSocial
53
54     // devuelve representación String de un objeto Empleado
55     public String toString()
56     {
57        return String.format( "%s %s\nnumero de seguro social: %s",
58           obtenerPrimerNombre(), obtenerApellidoPaterno(), obtenerNumeroSeguroSocial() );
59     } // fin del método toString
60
61     // Nota: Aquí no implementamos el método obtenerMontoPago de PorPagar, así que
62     // esta clase debe declararse como abstract para evitar un error de compilación.
63  } // fin de la clase abstracta Empleado
```

Figura 10.13 | La clase `Empleado`, que implementa a `PorPagar`. (Parte 2 de 2).

En la figura 10.13, manejamos esta situación en forma distinta. Recuerde que cuando una clase implementa a una interfaz, hace un contrato con el compilador, en el que se establece que la clase implementará cada uno de los métodos en la interfaz, o de lo contrario la clase se declara como `abstract`. Si se elige la última opción, no necesitamos declarar los métodos de la interfaz como `abstract` en la clase abstracta; ya están declarados como tales de manera implícita en la interfaz. Cualquier subclase concreta de la clase abstracta debe implementar a los métodos de la interfaz para cumplir con el contrato de la superclases con el compilador. Si la subclase no lo hace, también debe declararse como `abstract`. Como lo indican los comentarios en las líneas 61 y 62, la clase `Empleado` de la figura 10.13 no implementa al método `obtenerMontoPago`, por lo que la clase se declara como `abstract`. Cada subclase directa de `Empleado` hereda el contrato de la superclase para implementar el método

`obtenerMontoPago` y, por ende, debe implementar este método para convertirse en una clase concreta, para la cual puedan crearse instancias de objetos. Una clase que extienda a una de las subclases concretas de `Empleado` heredará una implementación de `obtenerMontoPago` y, por ende, también será una clase concreta.

10.7.5 Modificación de la clase `EmpleadoAsalariado` para usarla en la jerarquía `PorPagar`

La figura 10.14 contiene una versión modificada de la clase `EmpleadoAsalariado`, que extiende a `Empleado` y cumple con el contrato de la superclase `Empleado` para implementar el método `obtenerMontoPago` de la interfaz `PorPagar`. Esta versión de `EmpleadoAsalariado` es idéntica a la de la figura 10.5, con la excepción de que esta versión implementa al método `obtenerMontoPago` (líneas 30 a 33) en vez del método `ingresos`. Los dos métodos contienen la misma funcionalidad, pero tienen distintos nombres. Recuerde que la versión de `PorPagar` del método tiene un nombre más general para que pueda aplicarse a clases que sean posiblemente dispares. El resto de las subclases de `Empleado` (`EmpleadoPorHoras`, `EmpleadoPorComision` y `EmpleadoBaseMasComision`) también deben modificarse para que contengan el método `obtenerMontoPago` en vez de `ingresos`, y así reflejar el hecho de que ahora `Empleado` implementa a `PorPagar`. Dejaremos estas modificaciones como un ejercicio y sólo utilizaremos a `EmpleadoAsalariado` en nuestro programa de prueba en esta sección.

```
1  // Fig. 10.14: EmpleadoAsalariado.java
2  // La clase EmpleadoAsalariado extiende a Empleado, que implementa a PorPagar.
3
4  public class EmpleadoAsalariado extends Empleado
5  {
6     private double salarioSemanal;
7
8     // constructor con cuatro argumentos
9     public EmpleadoAsalariado( String nombre, String apellido, String nss,
10       double salario )
11    {
12       super( nombre, apellido, nss ); // pasa argumentos al constructor de Empleado
13       establecerSalarioSemanal( salario ); // valida y almacena el salario
14    } // fin del constructor de EmpleadoAsalariado con cuatro argumentos
15
16    // establece el salario
17    public void establecerSalarioSemanal( double salario )
18    {
19       salarioSemanal = salario < 0.0 ? 0.0 : salario;
20    } // fin del método establecerSalarioSemanal
21
22    // devuelve el salario
23    public double obtenerSalarioSemanal()
24    {
25       return salarioSemanal;
26    } // fin del método obtenerSalarioSemanal
27
28    // calcula los ingresos; implementa el método de la interfaz PorPagar
29    // que era abstracto en la superclase Empleado
30    public double obtenerMontoPago()
31    {
32       return obtenerSalarioSemanal();
33    } // fin del método obtenerMontoPago
34
35    // devuelve representación String de un objeto EmpleadoAsalariado
36    public String toString()
37    {
```

Figura 10.14 | La clase `EmpleadoAsalariado`, que implementa el método `obtenerMontoPago` de la interfaz `PorPagar`. (Parte 1 de 2).

```
      return String.format( "empleado asalariado: %s\n%s: $%,.2f",
         super.toString(), "salario semanal", obtenerSalarioSemanal() );
   } // fin del método toString
} // fin de la clase EmpleadoAsalariado
```

Figura 10.14 | La clase `EmpleadoAsalariado`, que implementa el método `obtenerMontoPago` de la interfaz `PorPaga`. (Parte 2 de 2).

Cuando una clase implementa a una interfaz, se aplica la misma relación *"es un"* que proporciona la herencia. Por ejemplo, la clase `Empleado` implementa a `PorPagar`, por lo que podemos decir que un objeto `Empleado` *es un* objeto `PorPagar`. De hecho, los objetos de cualquier clase que extienda a `Empleado` son también objetos `PorPagar`. Por ejemplo, los objetos `EmpleadoAsalariado` son objetos `PorPagar`. Al igual que con las relaciones de herencia, un objeto de una clase que implemente a una interfaz puede considerarse como un objeto del tipo de la interfaz. Los objetos de cualquier subclase de la clase que implementa a la interfaz también pueden considerarse como objetos del tipo de la interfaz. Por ende, así como podemos asignar la referencia de un objeto `EmpleadoAsalariado` a una variable de la superclase `Empleado`, también podemos asignar la referencia de un objeto `EmpleadoAsalariado` a una variable de la interfaz `PorPagar`. `Factura` implementa a `PorPagar`, por lo que un objeto `Factura` también *es un* objeto `PorPagar`, y podemos asignar la referencia de un objeto `Factura` a una variable `PorPagar`.



Observación de ingeniería de software 10.7

La herencia y las interfaces son similares en cuanto a su implementación de la relación "es un". *Un objeto de una clase que implementa a una interfaz puede considerarse como un objeto del tipo de esa interfaz. Un objeto de cualquier subclase de una clase que implemente a una interfaz también puede considerarse como un objeto del tipo de la interfaz.*



Observación de ingeniería de software 10.8

La relación "es un" *que existe entre las superclases y las subclases, y entre las interfaces y las clases que las implementan, se mantiene cuando se pasa un objeto a un método. Cuando el parámetro de un método recibe una variable de una superclase o de un tipo de interfaz, el método procesa en forma polimórfica al objeto que recibe como argumento.*



Observación de ingeniería de software 10.9

Al utilizar una referencia a la superclase, podemos invocar de manera polimórfica a cualquier método especificado en la declaración de la superclase (y en la clase `Object`*). Al utilizar una referencia a la interfaz, podemos invocar de manera polimórfica a cualquier método especificado en la declaración de la interfaz (y en la clase* `Object`*; ya que una variable de un tipo de interfaz debe hacer referencia a un objeto para llamar a los métodos, y todos los objetos contienen los métodos de la clase* `Object`*).*

10.7.6 Uso de la interfaz `PorPagar` para procesar objetos `Factura` y `Empleado` mediante el polimorfismo

`PruebaInterfazPorPagar` (figura 10.15) ilustra que la interfaz `PorPagar` puede usarse para procesar un conjunto de objetos `Factura` y `Empleado` en forma polimórfica en una sola aplicación. La línea 9 declara a `objetosPorPagar` y le asigna un arreglo de cuatro variables `PorPagar`. Las líneas 12 y 13 asignan las referencias de objetos `Factura` a los primeros dos elementos de `objetosPorPagar`. Después, las líneas 14 a 17 asignan las referencias de objetos `EmpleadoAsalariado` a los dos elementos restantes de `objetosPorPagar`. Estas asignaciones se permiten debido a que un objeto `Factura` *es un* objeto `PorPagar`, un `EmpleadoAsalariado` *es un* `Empleado`, y un `Empleado` *es un* objeto `PorPagar`. Las líneas 23 a 29 utilizan una instrucción `for` mejorada para procesar cada objeto `PorPagar` en `objetosPorPagar` de manera polimórfica, imprimiendo en pantalla el objeto como un `String`, junto con el pago vencido. Observe que la línea 27 invoca al método `toString` desde una referencia de la interfaz `PorPagar`, aun cuando `toString` no se declara en la interfaz `PorPagar`; todas las referencias (incluyendo las de los tipos de interfaces) se refieren a objetos que extienden a `Object` y, por lo tanto, tienen un método

```java
// Fig. 10.15: PruebaInterfazPorPagar.java
// Prueba la interfaz PorPagar.

public class PruebaInterfazPorPagar
{
   public static void main( String args[] )
   {
      // crea arreglo PorPagar con cuatro elementos
      PorPagar objetosPorPagar[] = new PorPagar[ 4 ];

      // llena el arreglo con objetos que implementan la interfaz PorPagar
      objetosPorPagar[ 0 ] = new Factura( "01234", "asiento", 2, 375.00 );
      objetosPorPagar[ 1 ] = new Factura( "56789", "llanta", 4, 79.95 );
      objetosPorPagar[ 2 ] =
         new EmpleadoAsalariado( "John", "Smith", "111-11-1111", 800.00 );
      objetosPorPagar[ 3 ] =
         new EmpleadoAsalariado( "Lisa", "Barnes", "888-88-8888", 1200.00 );

      System.out.println(
         "Facturas y Empleados procesados en forma polimorfica:\n" );

      // procesa en forma genérica cada elemento en el arreglo objetosPorPagar
      for ( PorPagar porPagarActual : objetosPorPagar )
      {
          // imprime porPagarActual y su monto de pago apropiado
         System.out.printf( "%s \n%s: $%,.2f\n\n",
            porPagarActual.toString(),
            "pago vencido", porPagarActual.obtenerMontoPago() );
      } // fin de for
   } // fin de main
} // fin de la clase PruebaInterfazPorPagar
```

```
Facturas y Empleados procesados en forma polimorfica:

factura:
numero de pieza: 01234 (asiento)
cantidad: 2
precio por articulo: $375.00
pago vencido: $750.00

factura:
numero de pieza: 56789 (llanta)
cantidad: 4
precio por articulo: $79.95
pago vencido: $319.80

empleado asalariado: John Smith
numero de seguro social: 111-11-1111
salario semanal: $800.00
pago vencido: $800.00

empleado asalariado: Lisa Barnes
numero de seguro social: 888-88-8888
salario semanal: $1,200.00
pago vencido: $1,200.00
```

Figura 10.15 | Programa de prueba para la interfaz `PorPagar`, que procesa objetos `Factura` y `Empleado` de manera polimórfica.

toString. (Observe que aquí también podemos invocar a toString en forma implícita). La línea 28 invoca al método obtenerMontoPago de PorPagar para obtener el monto a pagar para cada objeto en objetosPorPagar, sin importar el tipo actual del objeto. Los resultados revelan que las llamadas a los métodos en las líneas 27 y 28 invocan a la implementación de la clase apropiada de los métodos toString y obtenerMontoPago. Por ejemplo, cuando empleadoActual se refiere a un objeto Factura durante la primera iteración del ciclo for, se ejecutan los métodos toString y obtenerMontoPago de la clase Factura.



Observación de ingeniería de software 10.10

Todos los métodos de la clase Object *pueden llamarse mediante el uso de una referencia de un tipo de interfaz. Una referencia se refiere a un objeto, y todos los objetos heredan los métodos de la clase* Object*.*

10.7.7 Declaración de constantes con interfaces

Como mencionamos en la sección 10.7, una interfaz puede declarar constantes. De manera implícita, las constantes son public, static y final; de nuevo, no se requieren estas palabras en la declaración de la interfaz. Un uso popular de una interfaz es para declarar un conjunto de constantes que pueden utilizarse en muchas declaraciones de clases. Considere la siguiente interfaz Constantes:

```
public interface Constantes
{
    int UNO = 1;
    int DOS = 2;
    int TRES = 3;
}
```

Una clase puede usar estas constantes, para lo cual importa la interfaz y después hace referencia a cada constante como Constantes.UNO, Constantes.DOS y Constantes.TRES. Observe que una clase puede hacer referencia a las constantes importadas sólo con sus nombres (es decir, UNO, DOS y TRES) si utiliza una declaración static import (presentada en la sección 8.12) para importar la interfaz.



Observación de ingeniería de software 10.11

A partir de Java SE 5.0, una mejor práctica de programación fue crear conjuntos de constantes como enumeraciones, con la palabra clave enum*. En la sección 6.10 podrá consultar una introducción a* enum*, y en la sección 8.9 podrá ver los detalles adicionales sobre las* enums*.*

10.7.8 Interfaces comunes de la API de Java

En esta sección veremos las generalidades acerca de varias interfaces comunes que se encuentran en la API de Java. El poder y la flexibilidad de las interfaces se utilizan con frecuencia a lo largo de la API de Java. Estas interfaces se implementan y usan de la misma forma que las interfaces que usted crea (por ejemplo, la interfaz PorPagar en la sección 10.7.2). Como verá a lo largo de este libro, las interfaces de la API de Java le permiten utilizar sus propias clases dentro de los marcos de trabajo que proporciona Java, como el comparar objetos de sus propios tipos y crear tareas que se ejecuten de manera concurrente con otras tareas en el mismo programa. La figura 10.16 presenta una breve sinopsis de las interfaces más populares de la API de Java que utilizamos en este libro.

Interfaz	Descripción
Comparable	Como vimos en el capítulo 2, Java contiene varios operadores de comparación (<, <=, >, >=, ==, !=) que nos permiten comparar valores primitivos. Sin embargo, estos operadores no se pueden utilizar para comparar el contenido de los objetos. La interfaz Comparable se utiliza para permitir que los objetos de una clase que implementa a la interfaz se comparen entre sí. La interfaz contiene un método, compareTo, que compara el objeto que llama al método con el objeto que se pasa como argumento para el método. Las clases deben implementar

Figura 10.16 | Interfaces comunes de la API de Java. (Parte 1 de 2).

Interfaz	Descripción
`Comparable` (*continúa*)	a `compareTo` de tal forma que devuelva un valor indicando si el objeto en el cual se invoca es menor (valor de retorno entero negativo), igual (valor de retorno 0) o mayor (valor de retorno entero positivo) que el objeto que se pasa como argumento, utilizando cualquier criterio especificado por el programador. Por ejemplo, si la clase `Empleado` implementa a `Comparable`, su método `compareTo` podría comparar objetos `Empleado` en base a sus montos de ingresos. La interfaz `Comparable` se utiliza comúnmente para ordenar objetos en una colección como un arreglo. En el capítulo 18, Genéricos y en el capítulo 19, Colecciones, utilizaremos a `Comparable`.
`Serializable`	Una interfaz que se utiliza para identificar clases cuyos objetos pueden escribirse en (serializarse), o leerse desde (deserializarse) algún tipo de almacenamiento (archivo en disco, campo de base de datos) o transmitirse a través de una red. En el capítulo 14, Archivos y flujos y en el capítulo 24, Redes, utilizaremos a `Serializable`.
`Runnable`	La implementa cualquier clase para la cual sus objetos deban poder ejecutarse en paralelo, usando una técnica llamada subprocesamiento múltiple (que veremos en el capítulo 23, Subprocesamiento múltiple). La interfaz contiene un método, `run`, que describe el comportamiento de un objeto al ejecutarse.
Interfaces de escucha de eventos de la GUI	Usted trabaja con interfaces gráficas de usuario (GUI) a diario. Por ejemplo, en su navegador Web, podría escribir en un campo de texto la dirección de un sitio Web para visitarlo, o podría hacer clic en un botón para regresar al sitio anterior que visitó. Cuando escribe una dirección de un sitio Web o cuando hace clic en un botón en el navegador Web, éste debe responder a su interacción y realizar la tarea que usted le indica. Su interacción se conoce como evento, y el código que utiliza el navegador para responder a un evento se conoce como manejador de eventos. En el capítulo 11, Componentes de la GUI: parte 1, y en el capítulo 22, Componentes de la GUI: parte 2, aprenderá a crear GUIs en Java y cómo crear manejadores de eventos para responder a las interacciones del usuario. Los manejadores de eventos se declaran en clases que implementan una interfaz de escucha de eventos apropiada. Cada interfaz de escucha de eventos especifica uno o más métodos que deben implementarse para responder a las interacciones de los usuarios.
`SwingConstants`	Contiene un conjunto de constantes que se utilizan en la programación de GUI para posicionar los elementos de la GUI en la pantalla. En los capítulo 11 y 22 exploraremos la programación de GUI.

Figura 10.16 | Interfaces comunes de la API de Java. (Parte 2 de 2).

10.8 (Opcional) Ejemplo práctico de GUI y gráficos: realizar dibujos mediante el polimorfismo

Tal vez haya observado en el programa que creamos en el ejercicio 8.1 (y que modificamos en el ejercicio 9.1) que existen muchas similitudes entre las clases de figuras. Mediante la herencia, podemos "factorizar" las características comunes de las tres clases y colocarlas en una sola superclase de figura. Después, podemos manipular objetos de los tres tipos de figuras en forma polimórfica, usando variables del tipo de la superclase. Al eliminar la redundancia en el código se producirá un programa más pequeño y flexible, que será más fácil de mantener.

Ejercicios del ejemplo práctico de GUI y gráficos

10.1 Modifique las clases `Milinea`, `MiOvalo` y `MiRectangulo` de los ejercicios 8.1 y 9.1 para crear la jerarquía de clases de la figura 10.17. Las clases de la jerarquía `MiFigura` deben ser clases de figuras "inteligentes", que sepan cómo dibujarse a sí mismas (si se les proporciona un objeto `Graphics` que les indique en dónde deben dibujarse). Una vez que el programa cree un objeto a partir de esta jerarquía, podrá manipularlo de manera polimórfica por el resto de su duración como un objeto `MiFigura`.

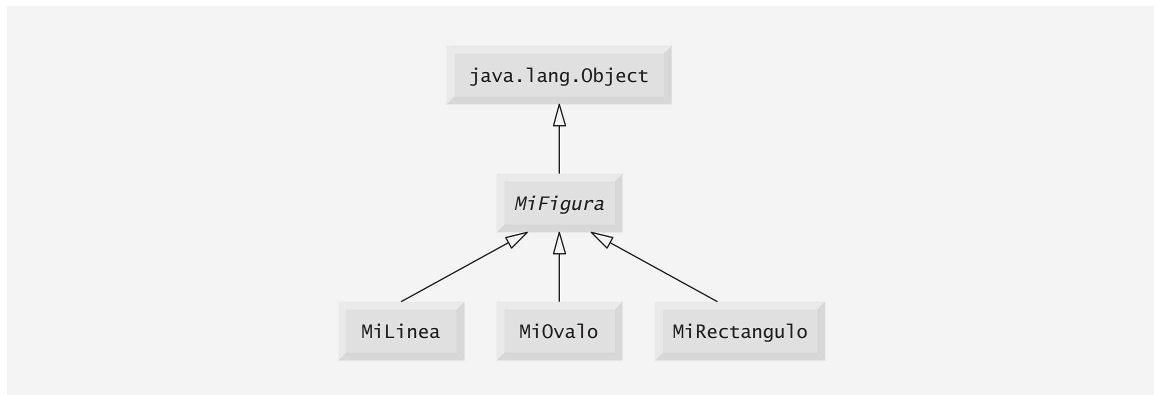


Figura 10.17 | La jerarquía `MiFigura`.

En su solución, la clase `MiFigura` en la figura 10.17 *debe* ser `abstract`. Como `MiFigura` representa a cualquier figura en general, no puede implementar un método `dibujar` sin saber exactamente qué figura es. Los datos que representan las coordenadas y el color de las figuras en la jerarquía deben declararse como miembros `private` de la clase `MiFigura`. Además de los datos comunes, la clase `MiFigura` debe declarar los siguientes métodos:

a) Un constructor sin argumentos que establezca todas las coordenadas de la figura en `0`, y el color en `Color.BLACK`.

b) Un constructor que inicialice las coordenadas y el color con los valores de los argumentos suministrados.

c) Métodos *establecer* para las coordenadas individuales y el color, que permitan al programador establecer cualquier pieza de datos de manera independiente, para una figura en la jerarquía.

d) Métodos *obtener* para las coordenadas individuales y el color, que permitan al programador obtener cualquier pieza de datos de manera independiente, para una figura en la jerarquía.

e) El método `abstract`

```
public abstract void dibujar( Graphics g );
```

que se llamará desde el método `paintComponent` del programa para dibujar una figura en la pantalla.

Para asegurar un correcto encapsulamiento, todos los datos en la clase `MiFigura` deben ser `private`. Para esto se requiere declarar métodos *establecer* y *obtener* para manipular los datos. La clase `Milinea` debe proporcionar un constructor sin argumentos y un constructor con argumentos para las coordenadas y el color. Las clases `MiOvalo` y `MiRectangulo` deben proporcionar un constructor sin argumentos y un constructor con argumentos para las coordenadas, el color y para determinar si la figura es rellena. El constructor sin argumentos debe, además, establecer los valores predeterminados, y la figura como una figura sin relleno.

Puede dibujar líneas, rectángulos y óvalos si conoce dos puntos en el espacio. Las líneas requieren coordenadas *x1*, *y1*, *x2* y *y2*. El método `drawLine` de la clase `Graphics` conectará los dos puntos suministrados con una línea. Si tiene los mismos cuatro valores de coordenadas (*x1*, *y1*, *x2* y *y2*) para óvalos y rectángulos, puede calcular los cuatro argumentos necesarios para dibujarlos. Cada uno requiere un valor de coordenada *x* superior izquierda (el menor de los dos valores de coordenada *x*), un valor de coordenada *y* superior izquierda (el menor de los dos valores de coordenada *y*), una *anchura* (el valor absoluto de la diferencia entre los dos valores de coordenada *x*) y una *altura* (el valor absoluto de la diferencia entre los dos valores de coordenada *y*). Los rectángulos y óvalos también deben tener una bandera `relleno`, que determine si se dibujará la figura con un relleno.

No debe haber variables `MiLinea`, `MiOvalo` o `MiRectangulo` en el programa; sólo variables `MiFigura` que contengan referencias a objetos `MiLinea`, `MiOvalo` y `MiRectangulo`. El programa debe generar figuras aleatorias y almacenarlas en un arreglo de tipo `MiFigura`. El método `paintComponent` debe recorrer el arreglo `MiFigura` y dibujar cada una de las figuras (es decir, mediante una llamada polimórfica al método `dibujar` de cada figura).

Permita al usuario que especifique (mediante un diálogo de entrada) el número de figuras a generar. Después, el programa generará y mostrará las figuras en pantalla, junto con una barra de estado para informar al usuario cuántas figuras de cada tipo se crearon.

10.2 *(Modificación de la aplicación de dibujo)* En el ejercicio 10.1, usted creó una jerarquía `MiFigura` en la cual las clases `MiLinea`, `MiOvalo` y `MiRectangulo` extienden a `MiFigura` directamente. Si su jerarquía estuviera diseñada apropiadamente, debería poder ver las similitudes entre las clases `MiOvalo` y `MiRectangulo`. Rediseñe y vuelva a implementar el código de las clases `MiOvalo` y `MiRectangulo`, para “factorizar” las características comunes en la clase abstracta `MiFiguraDelimitada`, para producir la jerarquía de la figura 10.18.

La clase `MiFiguraDelimitada` debe declarar dos constructores que imiten a los de `MiFigura`, sólo con un parámetro adicional para ver si la figura es rellena. La clase `MiFiguraDelimitada` también debe declarar métodos *obtener* y *establecer* para manipular la bandera de relleno y los métodos que calculan la coordenada *x* superior izquierda, la coordenada *y* superior izquierda, la anchura y la altura. Recuerde que los valores necesarios para dibujar un óvalo o un rectángulo se pueden calcular a partir de dos coordenadas (*x*, *y*). Si se diseñan de manera apropiada, las nuevas clases `MiOvalo` y `MiRectangulo` deberán tener dos constructores y un método `dibujar` cada una.

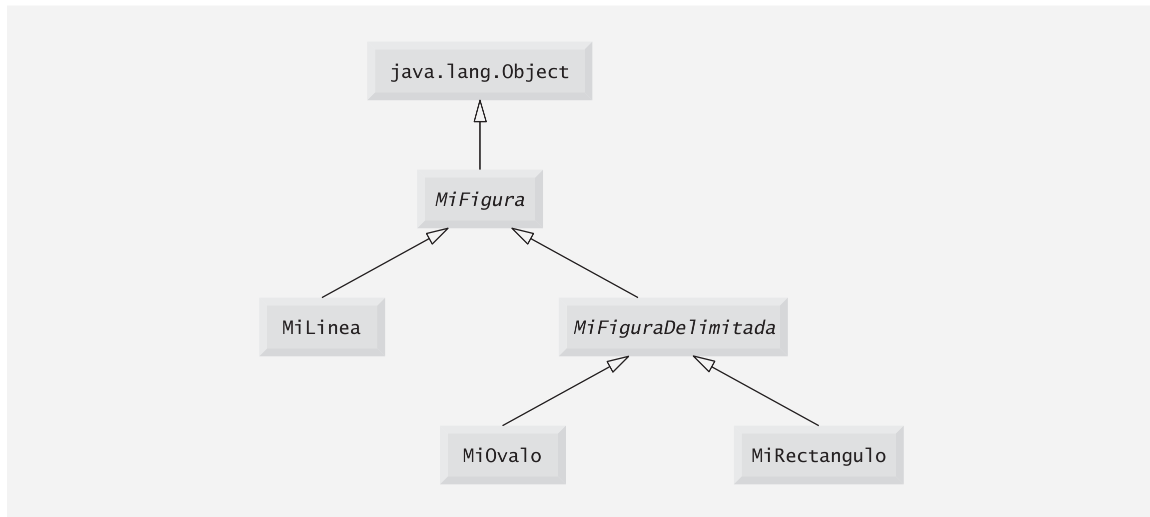


Figura 10.18 | Jerarquía `MiFigura` con `MiFiguraDelimitada`.

10.9 (Opcional) Ejemplo práctico de Ingeniería de Software: incorporación de la herencia en el sistema ATM

Ahora regresaremos a nuestro diseño del sistema ATM para ver cómo podría beneficiarse de la herencia. Para aplicar la herencia, primero buscamos características comunes entre las clases del sistema. Creamos una jerarquía de herencia para modelar las clases similares (pero no idénticas) en una forma elegante y eficiente. Después modificamos nuestro diagrama de clases para incorporar las nuevas relaciones de herencia. Por último, demostramos cómo traducir nuestro diseño actualizado en código de Java.

En la sección 3.10 nos topamos con el problema de representar una transacción financiera en el sistema. En vez de crear una clase para representar a todos los tipos de transacciones, optamos por crear tres clases distintas de transacciones (`SolicitudSaldo`, `Retiro` y `Deposito`) para representar las transacciones que puede realizar el sistema ATM. La figura 10.19 muestra los atributos y operaciones de las clases `SolicitudSaldo`, `Retiro` y `Deposito`. Observe que estas clases tienen un atributo (`numeroCuenta`) y una operación (`ejecutar`) en común. Cada clase requiere que el atributo `numeroCuenta` especifique la cuenta a la que se aplica la transacción. Cada clase contiene la operación `ejecutar`, que el ATM invoca para realizar la transacción. Es evidente que `SolicitudSaldo`, `Retiro` y `Deposito` representan *tipos de* transacciones. La figura 10.19 revela las características comunes entre las clases de transacciones, por lo que el uso de la herencia para factorizar las características comunes parece apropiado para diseñar estas clases. Colocamos la funcionalidad común en una superclase, `Transaccion`, que las clases `SolicitudSaldo`, `Retiro` y `Deposito` extienden.

UML especifica una relación conocida como **generalización** para modelar la herencia. La figura 10.20 es el diagrama de clases que modela la generalización de la superclase `Transaccion` y las subclases `SolicitudSaldo`,

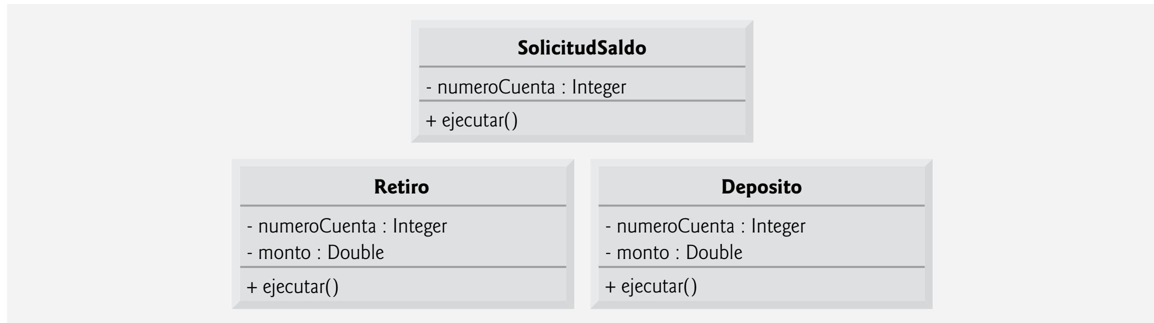


Figura 10.19 | Atributos y operaciones de las clases `SolicitudSaldo`, `Retiro` y `Deposito`.

`Retiro` y `Deposito`. Las flechas con puntas triangulares huecas indican que las clases `SolicitudSaldo`, `Retiro` y `Deposito` exienden a la clase `Transaccion`. Se dice que la clase `Transaccion` es una generalización de las clases `SolicitudSaldo`, `Retiro` y `Deposito`. Se dice que las clases `SolicitudSaldo`, `Retiro` y `Deposito` son **especializaciones** de la clase `Transaccion`.

Las clases `SolicitudSaldo`, `Retiro` y `Deposito` comparten el atributo entero `numeroCuenta`, por lo que factorizamos este atributo común y lo colocamos en la superclase `Transaccion`. Ya no listamos a `numeroCuenta` en el segundo compartimiento de cada subclase, ya que las tres subclases heredan este atributo de `Transaccion`. Sin embargo, recuerde que las subclases no pueden acceder a los atributos `private` de una superclase. Por lo tanto, incluimos el método `public obtenerNumeroCuenta` en la clase `Transaccion`. Cada subclase heredará este método, con lo cual podrá acceder a su `numeroCuenta` según sea necesario para ejecutar una transacción.

De acuerdo con la figura 10.19, las clases `SolicitudSaldo`, `Retiro` y `Deposito` también comparten la operación `ejecutar`, por lo que decidimos que la superclase `Transaccion` debe contener el método `public ejecutar`. Sin embargo, no tiene sentido implementar a `ejecutar` en la clase `Transaccion`, ya que la funcionalidad que proporciona este método depende del tipo específico de la transacción actual. Por lo tanto, declaramos el método `ejecutar` como `abstract` en la superclase `Transaccion`. Cualquier clase que contenga cuando menos un método abstracto también debe declararse como `abstract`. Esto obliga a que cualquier clase de `Transaccion` que deba ser una clase concreta (es decir, `SolicitudSaldo`, `Retiro` y `Deposito`) a implementar el método `ejecutar`. UML requiere que coloquemos los nombres de clase abstractos (y los métodos abstractos) cursivas, por lo cual `Transaccion` y su método `ejecutar` aparecen en cursivas en la figura 10.20. Observe que el método `ejecutar` no está en cursivas en las subclases `SolicitudSaldo`, `Retiro` y `Deposito`. Cada subclase sobrescribe el método `ejecutar` de la superclase `Transaccion` con una implementación concreta que realiza los pasos apropiados para completar ese tipo de transacción. Observe que la figura 10.20 incluye la operación `ejecutar` en el tercer compartimiento de las clases `SolicitudSaldo`, `Retiro` y `Deposito`, ya que cada clase tiene una implementación concreta distinta del método sobrescrito.

Al incorporar la herencia, se proporciona al `ATM` una manera elegante de ejecutar todas las transacciones "en general". Por ejemplo, suponga que un usuario elige realizar una solicitud de saldo. El `ATM` establece una referencia `Transaccion` a un nuevo objeto de la clase `SolicitudSaldo`. Cuando el `ATM` utiliza su referencia `Transaccion` para invocar el método `ejecutar`, se hace una llamada a la versión de `ejecutar` de `SolicitudSaldo`.

Este enfoque polimórfico también facilita la extensibilidad del sistema. Si deseamos crear un nuevo tipo de transacción (por ejemplo, una transferencia de fondos o el pago de un recibo), sólo tenemos que crear una subclase de `Transaccion` adicional que sobrescriba el método `ejecutar` con una versión apropiada para ejecutar el nuevo tipo de transacción. Sólo tendríamos que realizar pequeñas modificaciones al código del sistema, para permitir que los usuarios seleccionen el nuevo tipo de transacción del menú principal y para que la clase `ATM` cree instancias y ejecute objetos de la nueva subclase. La clase `ATM` podría ejecutar transacciones del nuevo tipo utilizando el código actual, ya que éste ejecuta todas las transacciones de manera polimórfica, usando una referencia `Transaccion` general.

Como aprendió antes en este capítulo, una clase abstracta como `Transaccion` es una para la cual el programador nunca tendrá la intención de crear instancias de objetos. Una clase abstracta sólo declara los atributos y comportamientos comunes de sus subclases en una jerarquía de herencia. La clase `Transaccion` define el concepto de lo que significa ser una transacción que tiene un número de cuenta y puede ejecutarse. Tal vez usted se

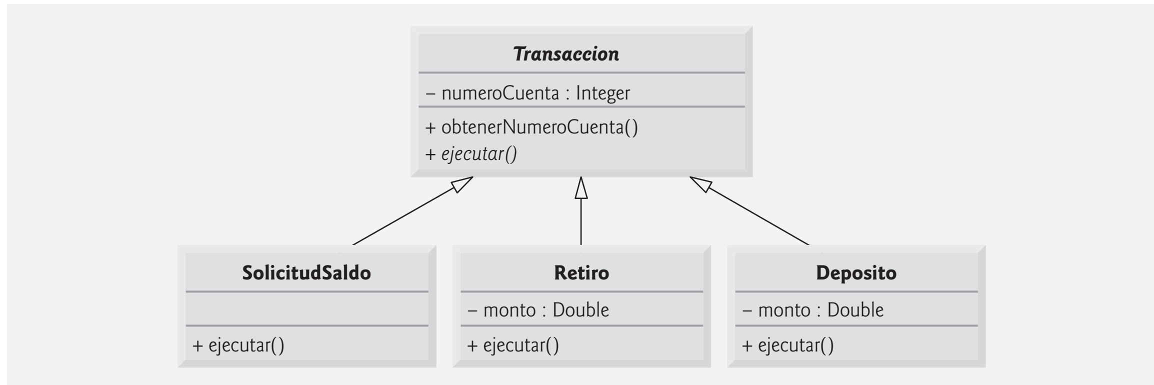


Figura 10.20 | Diagrama de clases que modela la generalización de la superclase `Transaccion` y las subclases `SolicitudSaldo`, `Retiro` y `Deposito`. Observe que los nombres de las clases abstractas (por ejemplo, `Transaccion`) y los nombres de los métodos (por ejemplo, `ejecutar` en la clase `Transaccion`) aparece en cursivas.

pregunte por qué nos tomamos la molestia de incluir el método `abstract ejecutar` en la clase `Transaccion`, si carece de una implementación concreta. En concepto, incluimos este método porque corresponde al comportamiento que define a todas las transacciones: ejecutarse. Técnicamente, debemos incluir el método `ejecutar` en la superclase `Transaccion`, de manera que la clase `ATM` (o cualquier otra clase) pueda invocar mediante el polimorfismo a la versión sobrescrita de este método en cada subclase, a través de una referencia `Transaccion`. Además, desde la perspectiva de la ingeniería de software, al incluir un método abstracto en una superclase, el que implementa las subclases se ve obligado a sobrescribir ese método con implementaciones concretas en las subclases, o de lo contrario, las subclases también serán abstractas, lo cual impedirá que se creen instancias de objetos de esas subclases.

Las subclases `SolicitudSaldo`, `Retiro` y `Deposito` heredan el atributo `numeroCuenta` de la superclase `Transaccion`, pero las clases `Retiro` y `Deposito` contienen el atributo adicional `monto` que las diferencia de la clase `SolicitudSaldo`. Las clases `Retiro` y `Deposito` requieren este atributo adicional para almacenar el monto de dinero que el usuario desea retirar o depositar. La clase `SolicitudSaldo` no necesita dicho atributo, puesto que sólo requiere un número de cuenta para ejecutarse. Aun cuando dos de las tres subclases de `Transaccion` comparten el atributo `monto`, no lo colocamos en la superclase `Transaccion`; en la superclase sólo colocamos las características comunes para todas las subclases, ya que de otra forma las subclases podrían heredar atributos (y métodos) que no necesitan y no deben tener.

La figura 10.21 presenta un diagrama de clases actualizado de nuestro modelo, en el cual se incorpora la herencia y se introduce la clase `Transaccion`. Modelamos una asociación entre la clase `ATM` y la clase `Transaccion` para mostrar que la clase `ATM`, en cualquier momento dado, está ejecutando una transacción o no lo está (es decir, existen cero o un objetos de tipo `Transaccion` en el sistema, en un momento dado). Como un `Retiro` es un tipo de `Transaccion`, ya no dibujamos una línea de asociación directamente entre la clase `ATM` y la clase `Retiro`. La subclase `Retiro` hereda la asociación de la superclase `Transaccion` con la clase `ATM`. Las subclases `SolicitudSaldo` y `Deposito` también heredan esta asociación, por lo que ya no existen las asociaciones entre la clase `ATM` y las clases `SolicitudSaldo` y `Deposito`, que se habían omitido anteriormente.

También agregamos una asociación entre la clase `Transaccion` y la clase `BaseDatosBanco` (figura 10.21). Todos los objetos `Transaccion` requieren una referencia a `BaseDatosBanco`, de manera que puedan acceder a (y modificar) la información de las cuentas. Debido a que cada subclase de `Transaccion` hereda esta referencia, ya no tenemos que modelar la asociación entre la clase `Retiro` y `BaseDatosBanco`. De manera similar, ya no existen las asociaciones entre `BaseDatosBanco` y las clases `SolicitudSaldo` y `Deposito`, que omitimos anteriormente.

Mostramos una asociación entre la clase `Transaccion` y la clase `Pantalla`. Todos los objetos `Transaccion` muestran los resultados al usuario a través de la `Pantalla`. Por ende, ya no incluimos la asociación que modelamos antes entre `Retiro` y `Pantalla`, aunque `Retiro` aún participa en las asociaciones con `DispensadorEfectivo` y `Teclado`. Nuestro diagrama de clases que incorpora la herencia también modela a `Deposito` y `SolicitudSaldo`. Mostramos las asociaciones entre `Deposito` y tanto `RanuraDeposito` como `Teclado`. Observe

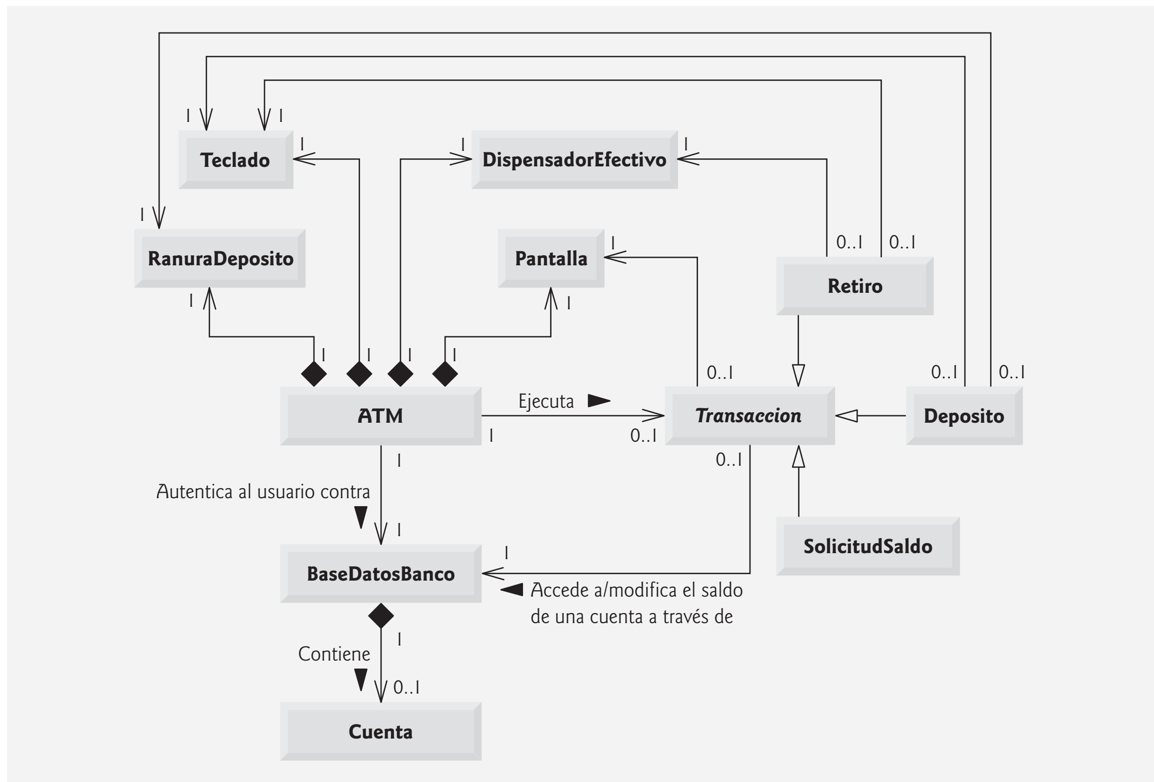


Figura 10.21 | Diagrama de clases del sistema ATM (en el que se incorpora la herencia). Observe que los nombres de las clases abstractas (`Transaccion`) aparecen en cursivas.

que la clase `SolicitudSaldo` no participa en asociaciones más que las heredadas de la clase `Transaccion`; un objeto `SolicitudSaldo` sólo necesita interactuar con la `BaseDatosBanco` y con la `Pantalla`.

El diagrama de clases de la figura 8.24 muestra los atributos y las operaciones con marcadores de visibilidad. Ahora presentamos un diagrama de clases modificado que incorpora la herencia en la figura 10.22. Este diagrama abreviado no muestra las relaciones de herencia, sino los atributos y los métodos después de haber empleado la herencia en nuestro sistema. Para ahorrar espacio, como hicimos en la figura 4.24, no incluimos los atributos mostrados por las asociaciones en la figura 10.21; sin embargo, los incluimos en la implementación en Java que aparece en el apéndice M. También omitimos todos los parámetros de las operaciones, como hicimos en la figura 8.24; al incorporar la herencia no se afectan los parámetros que ya estaban modelados en las figuras 6.22 a 6.25.



Observación de ingeniería de software 10.12

Un diagrama de clases completo muestra todas las asociaciones entre clases, junto con todos los atributos y operaciones para cada clase. Cuando el número de atributos, métodos y asociaciones de las clases es substancial (como en las figuras 10.21 y 10.22), una buena práctica que promueve la legibilidad es dividir esta información entre dos diagramas de clases: uno que se enfoque en las asociaciones y el otro en los atributos y métodos.

Implementación del diseño del sistema ATM (en el que se incorpora la herencia)

En la sección 8.19 empezamos a implementar el diseño del sistema ATM en código de Java. Ahora modificaremos nuestra implementación para incorporar la herencia, usando la clase `Retiro` como ejemplo.

1. Si la clase `A` es una generalización de la clase `B`, entonces la clase `B` extiende a la clase `A` en la declaración de la clase. Por ejemplo, la superclase abstracta `Transaccion` es una generalización de la clase `Retiro`. La figura 10.23 contiene la estructura de la clase `Retiro`, la cual contiene la declaración de clase apropiada.

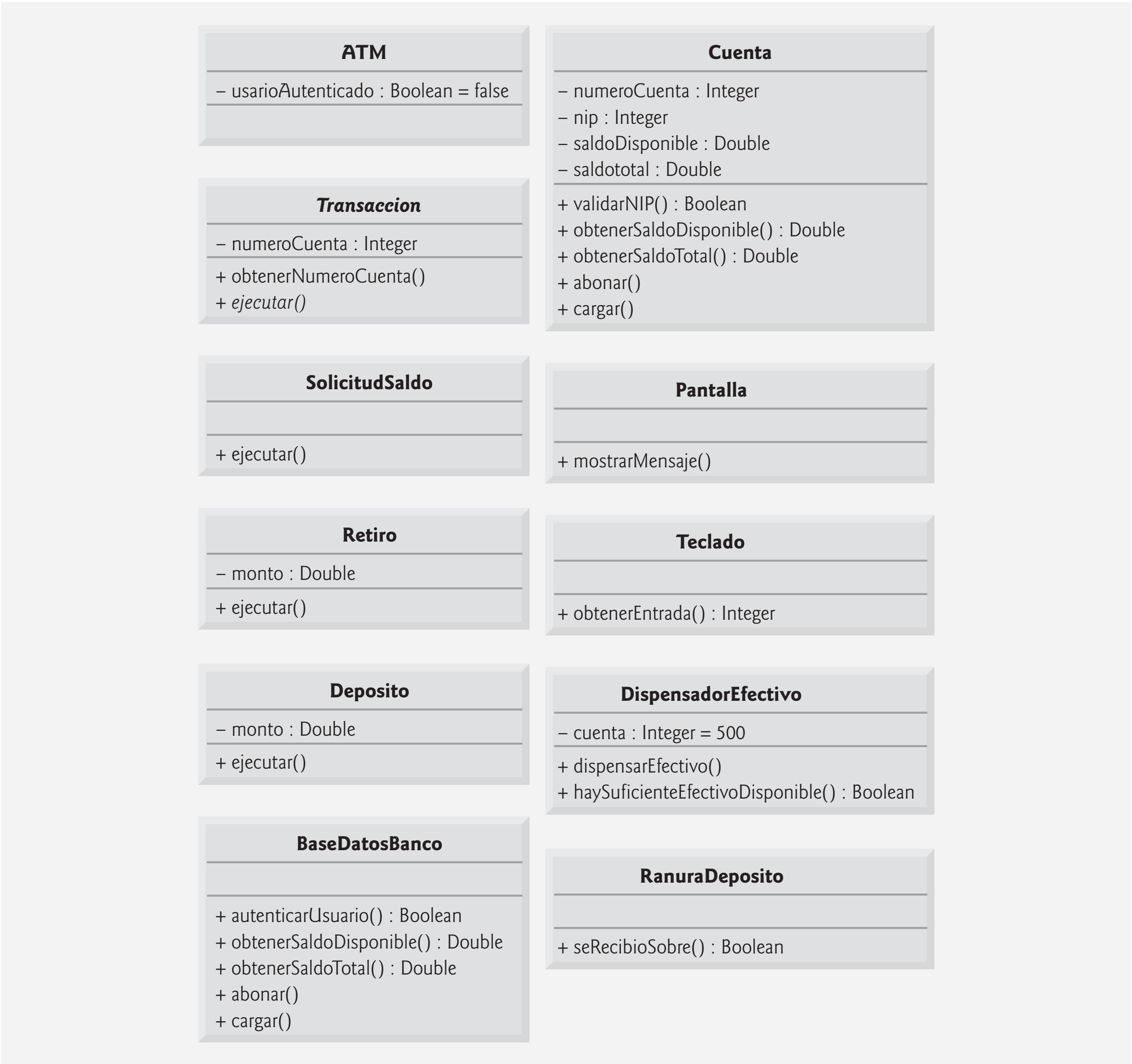


Figura 10.22 | Diagrama de clases con atributos y operaciones (incorporando la herencia). Observe que los nombres de las clases abstractas (`Transaccion`) y los nombres de los métodos (como `ejecutar` en la clase `Transaccion`) aparecen en cursiva.

```
// La clase Retiro representa una transacción de retiro en el ATM
public class Retiro extends Transaccion
{
}  // fin de la clase Retiro
```

Figura 10.23 | Código de Java para la estructura de la clase `Retiro`.

2. Si la clase `A` es una clase abstracta y la clase `B` es una subclase de la clase `A`, entonces la clase `B` debe implementar los métodos abstractos de la clase `A`, si la clase `B` va a ser una clase concreta. Por ejemplo, la clase `Transaccion` contiene el método abstracto `ejecutar`, por lo que la clase `Retiro` debe implementar este método si queremos crear una instancia de un objeto `Retiro`. La figura 10.24 es el código en Java para la clase `Retiro` de las figuras 10.21 y 10.22. La clase `Retiro` hereda el campo `numeroCuenta` de la

superclase `Transaccion`, por lo que `Retiro` no necesita declarar este campo. La clase `Retiro` también hereda referencias a las clases `Pantalla` y `BaseDatosBanco` de su superclase `Transaccion`, por lo que no incluimos estas referencias en nuestro código. La figura 10.22 especifica el atributo `monto` y la operación `ejecutar` para la clase `Retiro`. La línea 6 de la figura 10.24 declara un campo para el atributo `monto`. Las líneas 16 a 18 declaran la estructura de un método para la operación `ejecutar`. Recuerde que la subclase `Retiro` debe proporcionar una implementación concreta del método `abstract ejecutar` de la superclase `Transaccion`. Las referencias `teclado` y `dispensadorEfectivo` (líneas 7 y 8) son campos derivados de las asociaciones de `Retiro` en la figura 10.21. [*Nota:* el constructor en la versión completa de esta clase inicializará estas referencias con objetos reales].



Observación de ingeniería de software 10.13

Varias herramientas de modelado de UML convierten los diseños basados en UML en código de Java, y pueden agilizar el proceso de implementación en forma considerable. Para obtener más información sobre estas herramientas, consulte los recursos Web que se listan al final de la sección 2.9.

¡Felicidades por haber completado la porción correspondiente al diseño del ejemplo práctico! Esto concluye nuestro diseño orientado a objetos del sistema ATM. En el apéndice M implementamos por completo el sistema ATM, en 670 líneas de código en Java. Le recomendamos leer con cuidado el código y su descripción; ya que contiene muchos comentarios y sigue con precisión el diseño, con el cual usted ya está familiarizado. La descripción que lo acompaña está escrita cuidadosamente, para guiar su comprensión acerca de la implementación con base en el diseño de UML. Dominar este código es un maravilloso logro culminante, después de estudiar los capítulos 1 a 8.

```
1  // Retiro.java
2  // Se generó usando los diagramas de clases en las figuras 10.21 y 10.22
3  public class Retiro extends Transaccion
4  {
5     // atributos
6     private double monto;  // monto a retirar
7     private Teclado teclado;  // referencia al teclado
8     private DispensadorEfectivo dispensadorEfectivo;  // referencia al dispensador de
      efectivo
9
10    // constructor sin argumentos
11    public Retiro()
12    {
13    }  // fin del constructor de Retiro sin argumentos
14
15    // método que sobrescribe a ejecutar
16    public void ejecutar()
17    {
18    }  // fin del método ejecutar
19 } // fin de la clase Retiro
```

Figura 10.24 | Código de Java para la clase `Retiro`, basada en las figuras 10.21 y 10.22.

Ejercicios de autoevaluación del Ejemplo práctico de Ingeniería de Software

10.1 UML utiliza una flecha con una ________________ para indicar una relación de generalización.

a) punta con relleno sólido
b) punta triangular sin relleno
c) punta hueca en forma de diamante
d) punta lineal

10.2 Indique si el siguiente enunciado es *verdadero* o *falso* y, si es *falso*, explique por qué: UML requiere que subrayemos los nombres de las clases abstractas y los nombres de los métodos abstractos.

10.3 Escriba código en Java para empezar a implementar el diseño para la clase `Transaccion` que se especifica en las figuras 10.21 y 10.22. Asegúrese de incluir los atributos tipo referencias `private`, con base en las asociaciones de la clase `Transaccion`. Asegúrese también de incluir los métodos *establecer* `public` que proporcionan acceso a cualquiera de estos atributos `private` que requieren las subclases para realizar sus tareas.

Respuestas a los ejercicios de autoevaluación del Ejemplo práctico de Ingeniería de Software

10.1 b.

10.2 Falso. UML requiere que se escriban los nombres de las clases abstractas y de los métodos abstractos en cursiva.

10.3 El diseño para la clase `Transaccion` produce el código de la figura 10.25. Los cuerpos del constructor de la clase y los métodos se completarán en el apéndice M. Cuando estén implementados por completo, los métodos `obtenerPantalla` y `obtenerBaseDatosBanco` devolverán los atributos de referencias `private` de la superclase `Transaccion`, llamados `pantalla` y `baseDatosBanco`, respectivamente. Estos métodos permiten que las subclases de `Transaccion` accedan a la pantalla del ATM e interactúen con la base de datos del banco.

```java
// La clase abstracta Transaccion representa una transacción con el ATM
public abstract class Transaccion
{
   // atributos
   private int numeroCuenta; // indica la cuenta involucrada
   private Pantalla pantalla; // la pantalla del ATM
   private BaseDatosBanco baseDatosBanco; // base de datos de información de las cuentas

   // constructor sin argumentos invocado por las subclases, usando super()
   public Transaccion()
   {
   } // fin del constructor Transaccion sin argumentos

   // devuelve el número de cuenta
   public int obtenerNumeroCuenta()
   {
   } // fin del método obtenerNumeroCuenta

   // devuelve referencia a la pantalla
   public Pantalla obtenerPantalla()
   {
   } // fin del metodo getScreen

   // devuelve referencia a la base de datos del banco
   public BaseDatosBanco obtenerBaseDatosBanco()
   {
   } // fin del método obtenerBaseDatosBanco

   // método abstracto sobrescrito por las subclases
   public abstract void ejecutar();
} // fin de la clase Transaccion
```

Figura 10.25 | Código de Java para la clase `Transaccion`, basada en las figuras 10.21 y 10.22.

10.10 Conclusión

En este capítulo se introdujo el polimorfismo: la habilidad de procesar objetos que comparten la misma superclase en una jerarquía de clases, como si todos fueran objetos de la superclase. En este capítulo hablamos sobre cómo el polimorfismo facilita la extensibilidad y el mantenimiento de los sistemas, y después demostramos cómo utilizar

métodos sobrescritos para llevar a cabo el comportamiento polimórfico. Presentamos la noción de las clases abstractas, las cuales permiten a los programadores proporcionar una superclase apropiada, a partir de la cual otras clases pueden heredar. Aprendió que una clase abstracta puede declarar métodos abstractos que cada una de sus subclases debe implementar para convertirse en clase concreta, y que un programa puede utilizar variables de una clase abstracta para invocar implementaciones en las subclases de los métodos abstractos en forma polimórfica. También aprendió a determinar el tipo de un objeto en tiempo de ejecución. Por último, hablamos también sobre la declaración e implementación de una interfaz, como otra manera de obtener el comportamiento polimórfico.

Ahora deberá estar familiarizado con las clases, los objetos, el encapsulamiento, la herencia, las interfaces y el polimorfismo: los aspectos más esenciales de la programación orientada a objetos. En el siguiente capítulo analizaremos con más detalle las interfaces gráficas de usuario (GUIs).

Resumen

Sección 10.1 Introducción

- El polimorfismo nos permite escribir programas para procesar objetos que compartan la misma superclase en una jerarquía de clases, como si todos fueran objetos de la superclase; esto puede simplificar la programación.
- Con el polimorfismo, podemos diseñar e implementar sistemas que puedan extenderse con facilidad; pueden agregarse nuevas clases con sólo modificar un poco (o nada) las porciones generales del programa, siempre y cuando las nuevas clases sean parte de la jerarquía de herencia que el programa procesa en forma genérica. Las únicas partes de un programa que deben alterarse para dar cabida a las nuevas clases son las que requieren un conocimiento directo de las nuevas clases que el programador agregará a la jerarquía.

Sección 10.3 Demostración del comportamiento polimórfico

- Cuando el compilador encuentra una llamada a un método que se realiza a través de una variable, determina si el método puede llamarse verificando el tipo de clase de la variable. Si esa clase contiene la declaración del método apropiada (o hereda una), se compila la llamada. En tiempo de ejecución, el tipo del objeto al cual se refiere la variable es el que determina el método que se utilizará.

Sección 10.4 Clases y métodos abstractos

- En algunos casos es conveniente declarar clases para las cuales no tenemos la intención de crear instancias de objetos. A dichas clases se les conoce como clases abstractas. Como se utilizan sólo como superclases en jerarquías de herencia, nos referimos a ellas como superclases abstractas. Estas clases no pueden utilizarse para instanciar objetos, ya que están incompletas.
- El propósito principal de una clase abstracta es proporcionar una superclase apropiada, a partir de la cual puedan heredar otras clases y, por ende, compartir un diseño común.
- Las clases que pueden utilizarse para instanciar objetos se llaman clases concretas. Dichas clases proporcionan implementaciones de cada método que declaran (algunas de las implementaciones pueden heredarse).
- No todas las jerarquías de herencia contienen clases abstractas. Sin embargo, a menudo los programadores escriben código cliente que utiliza sólo tipos de superclases abstractas para reducir las dependencias del código cliente en un rango de tipos de subclases específicas.
- Algunas veces las clases abstractas constituyen varios niveles de la jerarquía.
- Para hacer una clase abstracta, ésta se declara con la palabra clave abstract. Por lo general, una clase abstracta contiene uno o más métodos abstractos.
- Los métodos abstractos no proporcionan implementaciones.
- Una clase que contiene métodos abstractos debe declararse como clase abstracta, aun si esa clase contiene métodos concretos (no abstractos). Cada subclase concreta de una superclase abstracta también debe proporcionar implementaciones concretas de los métodos abstractos de la superclase.
- Los constructores y los métodos `static` no pueden declararse como `abstract`.
- Aunque no podemos instanciar objetos de superclases abstractas, sí *podemos* usar superclases abstractas para declarar variables que puedan guardar referencias a objetos de cualquier clase concreta que se derive de esas superclases abstractas. Por lo general, los programas utilizan dichas variables para manipular los objetos de las subclases mediante el polimorfismo.
- En especial, el polimorfismo es efectivo para implementar los sistemas de software en capas.

Sección 10.5 Ejemplo práctico: sistema de nómina utilizando polimorfismo

- Al incluir un método abstracto en una superclase, obligamos a cada subclase directa de la superclase a sobrescribir ese método abstracto para que pueda convertirse en una clase concreta. Esto permite al diseñador de la jerarquía de clases demandar que cada subclase concreta proporcione una implementación apropiada del método.
- La mayoría de las llamadas a los métodos se resuelven en tiempo de ejecución, con base en el tipo del objeto que se está manipulando. Este proceso se conoce como vinculación dinámica o vinculación postergada.
- Una referencia a la superclase puede utilizarse para invocar sólo a métodos de la superclase (y la superclase puede invocar versiones sobrescritas de éstos en la subclase).
- El operador `instanceof` se puede utilizar para determinar si el tipo de un objeto específico tiene la relación *"es un"* con un tipo específico.
- Todos los objetos en Java conocen su propia clase y pueden acceder a esta información a través del método `getClass`, que todas las clases heredan de la clase `Object`. El método `getClass` devuelve un objeto de tipo `Class` (del paquete `java.lang`), el cual contiene información acerca del tipo del objeto, incluyendo el nombre de su clase.
- La relación *"es un"* se aplica sólo entre la subclase y sus superclases, no viceversa.
- No está permitido tratar de invocar a los métodos que sólo pertenecen a la subclase, en una referencia a la superclase. Aunque el método que se llame en realidad depende del tipo del objeto en tiempo de ejecución, podemos usar una variable para invocar sólo a los métodos que sean miembros del tipo de esa variable, que el compilador verifica.

Sección 10.6 Métodos y clases `final`

- Un método que se declara como `final` en una superclase no se puede redefinir en una subclase.
- Los métodos que se declaran como `private` son `final` de manera implícita, ya que es imposible sobrescribirlos en una subclase.
- Los métodos que se declaran como `static` son `final` de manera implícita.
- La declaración de un método `final` no puede cambiar, por lo que todas las subclases utilizan la misma implementación del método, y las llamadas a los métodos `final` se resuelven en tiempo de compilación; a esto se le conoce como vinculación estática.
- Como el compilador sabe que los métodos `final` no se pueden sobrescribir, puede optimizar los programas al eliminar las llamadas a los métodos `final` y sustituirlas con el código expandido de sus declaraciones en cada una de las ubicaciones de las llamadas al método; a esta técnica se le conoce como poner el código en línea.
- Una clase que se declara como `final` no puede ser una superclase (es decir, una clase no puede extender a una clase `final`).
- Todos los métodos en una clase `final` son implícitamente `final`.

Sección 10.7 Ejemplo práctico: creación y uso de interfaces

- Las interfaces definen y estandarizan las formas en que las cosas como las personas y los sistemas pueden interactuar entre sí.
- Una interfaz especifica *qué* operaciones están permitidas, pero no especifica *cómo* se realizan estas operaciones.
- Una interfaz de Java describe a un conjunto de métodos que pueden llamarse en un objeto.
- La declaración de una interfaz empieza con la palabra clave `interface` y sólo contiene constantes y métodos `abstract`.
- Todos los miembros de una interfaz deben ser `public`, y las interfaces no pueden especificar ningún detalle de implementación, como las declaraciones de métodos concretos y las variables de instancia.
- Todos los métodos que se declaran en una interfaz son `public abstract` de manera implícita, y todos los campos son `public`, `static` y `final` de manera implícita.
- Para utilizar una interfaz, una clase concreta debe especificar que implementa (`implements`) a esa interfaz, y debe declarar cada uno de los métodos de la interfaz con la firma especificada en su declaración. Una clase que no implementa a todos los métodos de una interfaz es una clase abstracta, por lo cual debe declararse como `abstract`.
- Implementar una interfaz es como firmar un contrato con el compilador que diga, "Declararé todos los métodos especificados por la interfaz, o declararé mi clase como `abstract`".
- Por lo general, una interfaz se utiliza cuando clases dispares (es decir, no relacionadas) necesitan compartir métodos y constantes comunes. Esto permite que los objetos de clases no relacionadas se procesen en forma polimórfica; los objetos de clases que implementan la misma interfaz pueden responder a las mismas llamadas a métodos.
- Usted puede crear una interfaz que describa la funcionalidad deseada, y después implementar esa interfaz en cualquier clase que requiera esa funcionalidad.
- A menudo, una interfaz se utiliza en vez de una clase `abstract` cuando no hay una implementación predeterminada que heredar; esto es, no hay campos ni implementaciones de métodos predeterminadas.

- Al igual que las clases `public abstract`, las interfaces son comúnmente de tipo `public`, por lo que se declaran en archivos por sí solas con el mismo nombre que la interfaz, y la extensión de archivo `.java`.
- Java no permite que las subclases hereden de más de una superclase, pero sí permite que una clase herede de una superclase e implemente más de una interfaz. Para implementar más de una interfaz, utilice una lista separada por comas de nombres de interfaz después de la palabra clave `implements` en la declaración de la clase.
- Todos los objetos de una clase que implementan varias interfaces tienen la relación *"es un"* con cada tipo de interfaz implementada.
- Una interfaz puede declarar constantes. Las constantes son implícitamente `public`, `static` y `final`.

Terminología

`abstract`, palabra clave
clase abstracta
clase concreta
clase iteradora
`Class`, clase
constantes declaradas en una interfaz
conversión descendente
declaración de una interfaz
especialización en UML
`final`, clase
`final`, método
generalización en UML
`getClass`, método de `Object`
`getName`, método de `Class`
herencia de implementación
herencia de interfaz
implementar una interfaz
`implements`, palabra clave
`instanceof`, operador
`interface`, palabra clave
método abstracto
polimorfismo
poner en línea llamadas a métodos
realización en UML
referencia a una subclase
referencia a una superclase
superclase abstracta
vinculación dinámica
vinculación estática
vinculación postergada

Ejercicios de autoevaluación

10.1 Complete las siguientes oraciones:

a) El polimorfismo ayuda a eliminar la lógica de ___________________.
b) Si una clase contiene al menos un método abstracto, es una clase ___________________.
c) Las clases a partir de las cuales pueden instanciarse objetos se llaman clases ___________________.
d) El ___________________ implica el uso de una variable de superclase para invocar métodos en objetos de superclase y subclase, lo cual nos permite "programar en general".
e) Los métodos que no son métodos de interfaz y que no proporcionan implementaciones deben declararse utilizando la palabra clave ___________________.
g) Al proceso de convertir una referencia almacenada en una variable de una superclase a un tipo de una subclase se le conoce como ___________________.

10.2 Conteste con *verdadero* o *falso* a cada una de las siguientes proposiciones; en caso de ser *falso*, explique por qué.

a) Es posible tratar a los objetos de superclase y a los objetos de subclase de manera similar.
b) Todos los métodos en una clase `abstract` deben declararse como métodos `abstract`.
c) Es peligroso tratar de invocar a un método que sólo pertenece a una subclase, a través de una variable de subclase.
d) Si una superclase declara a un método como `abstract`, una subclase debe implementar a ese método.
e) Un objeto de una clase que implementa a una interfaz puede considerarse como un objeto de ese tipo de interfaz.

Respuestas a los ejercicios de autoevaluación

10.1 a) `switch`. b) abstracta. c) concretas. d) polimorfismo. e) `abstract`. f) conversión descendente.

10.2 a) Verdadero. b) Falso. Una clase abstracta puede incluir métodos con implementaciones y métodos `abstract`. c) Falso. Es peligroso tratar de invocar un método que sólo pertenece a una subclase, con una variable de la superclase. d) Falso. Sólo una subclase concreta debe implementar el método. e) Verdadero.

Ejercicios

10.3 ¿Cómo es que el polimorfismo le permite programar "en forma general", en lugar de hacerlo "en forma específica"? Hable sobre las ventajas clave de la programación "en forma general".

10.4 Una subclase puede heredar la "interfaz" o "implementación" de una superclase. ¿En qué difieren las jerarquías de herencia diseñadas para heredar la interfaz, de las jerarquías diseñadas para heredar la implementación?

10.5 ¿Qué son los métodos abstractos? Describa las circunstancias en las que un método abstracto sería apropiado.

10.6 ¿Cómo es que el polimorfismo fomenta la extensibilidad?

10.7 Describa cuatro formas en las que podemos asignar referencias de superclases y subclases a variables de los tipos de las superclases y las subclases.

10.8 Compare y contraste las clases abstractas y las interfaces. ¿Para qué podría usar una clase abstracta? ¿Para qué podría usar una interfaz?

10.9 *(Modificación al sistema de nómina)* Modifique el sistema de nómina de las figuras 10.4 a 10.9 para incluir la variable de instancia `private` llamada `fechaNacimiento` en la clase `Empleado`. Use la clase `Fecha` de la figura 8.7 para representar el cumpleaños de un empleado. Agregue métodos obtener a la clase `Fecha` y sustituya el método `aStringFecha` con el método `toString`. Suponga que la nómina se procesa una vez al mes. Cree un arreglo de variables `Empleado` para guardar referencias a los diversos objetos empleado. En un ciclo, calcule la nómina para cada `Empleado` (mediante el polimorfismo) y agregue una bonificación de $100.00 a la cantidad de pago de nómina de la persona, si el mes actual es el mes en el que ocurre el cumpleaños de ese `Empleado`.

10.10 *(Jerarquía de figuras)* Implemente la jerarquía `Figura` que se muestra en la figura 9.3. Cada `FiguraBidimensional` debe contener el método `obtenerArea` para calcular el área de la figura bidimensional. Cada `FiguraTridimensional` debe tener los métodos `obtenerArea` y `obtenerVolumen` para calcular el área superficial y el volumen, respectivamente, de la figura tridimensional. Cree un programa que utilice un arreglo de referencias `Figura` a objetos de cada clase concreta en la jerarquía. El programa deberá imprimir una descripción de texto del objeto al cual se refiere cada elemento del arreglo. Además, en el ciclo que procesa a todas las figuras en el arreglo, determine si cada figura es `FiguraBidimensional` o `FiguraTridimensional`. Si es `FiguraBidimensional`, muestre su área. Si es `FiguraTridimensional`, muestre su área y su volumen.

10.11 *(Modificación al sistema de nómina)* Modifique el sistema de nómina de las figuras 10.4 a 10.9, para incluir una subclase adicional de `Empleado` llamada `TrabajadorPorPiezas`, que represente a un empleado cuyo sueldo se base en el número de piezas de mercancía producidas. La clase `TrabajadorPorPiezas` debe contener las variables de instancia `private` llamadas `sueldo` (para almacenar el sueldo del empleado por pieza) y `piezas` (para almacenar el número de piezas producidas). Proporcione una implementación concreta del método `ingresos` en la clase `TrabajadorPorPiezas` que calcule los ingresos del empleado, multiplicando el número de piezas producidas por el sueldo por pieza. Cree un arreglo de variables `Empleado` para almacenar referencias a objetos de cada clase concreta en la nueva jerarquía `Empleado`. Para cada `Empleado`, muestre su representación de cadena y los ingresos.

10.12 *(Modificación al sistema de cuentas por pagar)* En este ejercicio modificaremos la aplicación de cuentas por pagar de las figuras 10.11 a 10.15, para incluir la funcionalidad completa de la aplicación de nómina de las figuras 10.4 a 10.9. La aplicación debe aún procesar dos objetos `Factura`, pero ahora debe procesar un objeto de cada una de las cuatro subclases de `Empleado`. Si el objeto que se está procesando en un momento dado es `EmpleadoBasePorComision`, la aplicación debe incrementar el salario base del `EmpleadoBasePorComision` por un 10%. Por último, la aplicación debe imprimir el monto del pago para cada objeto. Complete los siguientes pasos para crear la nueva aplicación:

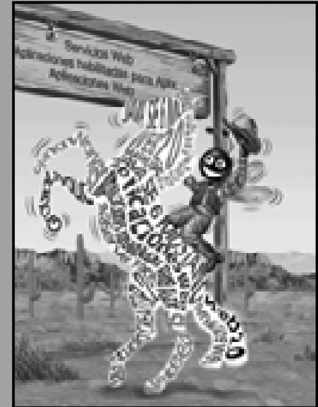
a) Modifique las clases `EmpleadoPorHoras` (figura 10.6) y `EmpleadoPorComision` (figura 10.7) para colocarlas en la jerarquía `PorPagar` como subclases de la versión de `Empleado` (figura 10.13) que implementa a `PorPagar`. [*Sugerencia:* cambie el nombre del método `ingresos` a `obtenerMontoPago` en cada subclase, de manera que la clase cumpla con su contrato heredado con la interfaz `PorPagar`].

b) Modifique la clase `EmpleadoBaseMasComision` (figura 10.8), de tal forma que extienda la versión de la clase `EmpleadoPorComision` que se creó en la parte *a*.

c) Modifique `PruebaInterfazPorPagar` (figura 10.15) para procesar mediante el polimorfismo dos objetos `Factura`, un `EmpleadoAsalariado`, un `EmpleadoPorHoras`, un `EmpleadoPorComision` y un `EmpleadoBaseMasComision`. Primero imprima una representación de cadena de cada objeto `PorPagar`. Después, si un objeto es un `EmpleadoBaseMasComision`, aumente su salario base por un 10%. Por último, imprima el monto del pago para cada objeto `PorPagar`.

11

Componentes de la GUI: parte 1



¿Crees que puedo escuchar todo el día esas cosas?

—Lewis Carroll

Inclusive, hasta un evento menor en la vida de un niño es un evento del mundo de ese niño y, por ende, es un evento del mundo.

—Gastón Bachelard

Tú pagas, por lo tanto, tú decides.

—Punch

Adivina si puedes, elige si te atreves.

—Pierre Corneille

OBJETIVOS

En este capítulo aprenderá a:

- Comprender los principios de diseño de las interfaces gráficas de usuario (GUI).
- Crear interfaces gráficas de usuario y manejar los eventos generados por las interacciones de los usuarios con las GUIs.
- Aprender acerca de los paquetes que contienen componentes relacionados con las GUIs, clases e interfaces manejadoras de eventos.
- Crear y manipular botones, etiquetas, listas, campos de texto y paneles.
- Entender el manejo de los eventos de ratón y los eventos de teclado.
- Aprender a utilizar los administradores de esquemas para ordenar los componentes de las GUIs.

Plan general

11.1 Introducción

Una **interfaz gráfica de usuario (GUI)** presenta un mecanismo amigable al usuario para interactuar con una aplicación. Una GUI proporciona a una aplicación una "apariencia visual" única. Al proporcionar distintas aplicaciones en las que los componentes de la interfaz de usuario sean consistentes e intuitivos, los usuarios pueden familiarizarse en cierto modo con una aplicación, de manera que pueden aprender a utilizarla en menor tiempo y con mayor productividad.



Observación de apariencia visual 11.1

Las interfaces de usuario consistentes permiten a un usuario aprender, con más rapidez, a utilizar nuevas aplicaciones.

Como ejemplo de una GUI, en la figura 11.1 se muestra una ventana del navegador Web Microsoft Internet Explorer, con algunos componentes de la GUI etiquetados. En la parte superior hay una **barra de título** que contiene el título de la ventana. Debajo de la barra de título hay una **barra de menús** que contiene **menús** (Archivo, Edición, Ver, etcétera). Debajo de la barra de menús hay un conjunto de **botones** que el usuario puede oprimir para realizar tareas en Internet Explorer. Debajo de los botones hay un **cuadro combinado**; donde el usuario puede escribir el nombre de un sitio Web a visitar, o hacer clic en la flecha hacia abajo, que se encuentra del lado derecho, para ver una lista de los sitios visitados previamente. Los menús, botones y el cuadro combinado son parte de la GUI de Internet Explorer. Éstos le permiten interactuar con el navegador Web.

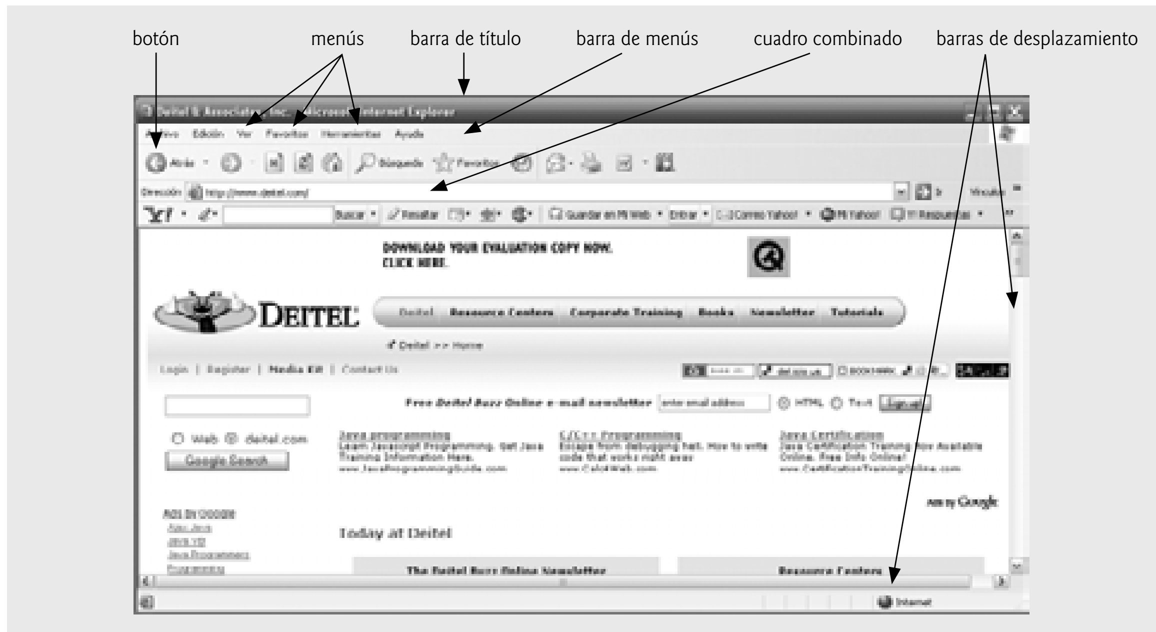


Figura 11.1 | Ventana de Microsoft Internet Explorer con sus componentes de la GUI.

Las GUIs se crean a partir de **componentes de la GUI**. A éstos se les conoce también como **controles** o **widgets** (**accesorios de ventana**) en otros lenguajes. Un componente de la GUI es un objeto con el cual interactúa el usuario mediante el ratón, el teclado u otra forma de entrada, como el reconocimiento de voz. En este capítulo y en el capítulo 22, Componentes de la GUI: parte 2, aprenderá acerca de muchos de los componentes de la GUI de Java. [*Nota:* varios conceptos que se cubren en este capítulo ya se han cubierto en el Ejemplo práctico opcional de GUI y gráficos de los capítulos 3 a 10. Por lo tanto, cierto material será repetido si usted leyó el ejemplo práctico. No necesita leer el ejemplo práctico para comprender este capítulo].

11.2 Entrada/salida simple basada en GUI con `JOptionPane`

Las aplicaciones en los capítulos 2 a 10 muestran texto en la ventana de comandos y obtienen la entrada de la ventana de comandos. La mayoría de las aplicaciones que usamos a diario utilizan ventanas o **cuadros de diálogo** (también conocidos como **diálogos**) para interactuar con el usuario. Por ejemplo, los programas de correo electrónico le permiten escribir y leer mensajes en una ventana que proporciona el programa. Por lo general, los cuadros de diálogo son ventanas en las cuales los programas muestran mensajes importantes al usuario, u obtienen información de éste. La clase **`JOptionPane`** de Java (paquete `javax.swing`) proporciona cuadros de diálogo preempaquetados para entrada y salida. Estos diálogos se muestran mediante la invocación de los métodos `static` de `JOptionPane`. La figura 11.2 presenta una aplicación simple de suma, que utiliza dos **diálogos de entrada** para obtener enteros del usuario, y un **diálogo de mensaje** para mostrar la suma de los enteros que introduce el usuario.

Diálogos de entrada

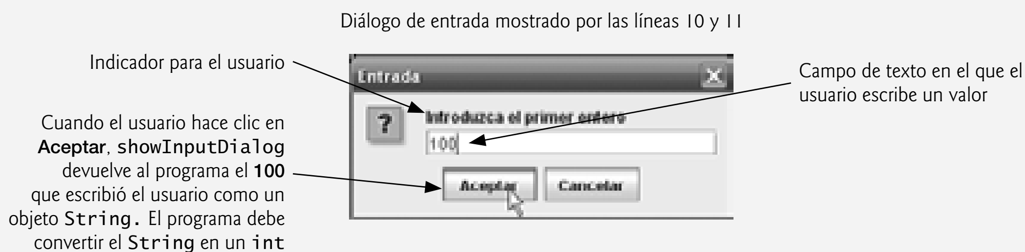
La línea 3 importa la clase `JOptionPane` para utilizarla en esta aplicación. Las líneas 10 y 11 declaran la variable `String` `primerNumero`, y le asignan el resultado de la llamada al método `static` **`showInputDialog`** de `JOptionPane`. Este método muestra un diálogo de entrada (vea la primera captura de pantalla en la figura 11.2), usando el argumento `String` (`"Introduzca el primer entero"`) como indicador.

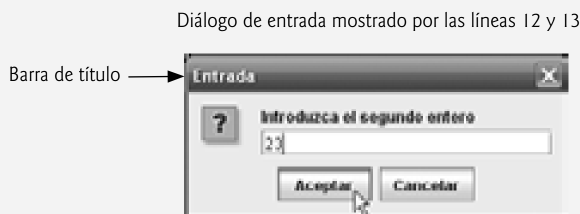


Observación de apariencia visual 11.2

El indicador en un diálogo de entrada utiliza comúnmente la ***capitalización estilo oración****: un estilo que capitaliza sólo la primera letra de la primera palabra en el texto, a menos que la palabra sea un nombre propio (por ejemplo, Deitel).*

```
1  // Fig. 11.2: Suma.java
2  // Programa de suma que utiliza a JOptionPane para entrada y salida.
3  import javax.swing.JOptionPane; // el programa usa JOptionPane
4
5  public class Suma
6  {
7     public static void main( String args[] )
8     {
9        // obtiene la entrada del usuario de los diálogos de entrada de JOptionPane
10       String primerNumero =
11          JOptionPane.showInputDialog( "Introduzca el primer entero" );
12       String segundoNumero =
13          JOptionPane.showInputDialog( "Introduzca el segundo entero" );
14
15       // convierte las entradas String en valores int para usarlos en un cálculo
16       int numero1 = Integer.parseInt( primerNumero );
17       int numero2 = Integer.parseInt( segundoNumero );
18
19       int suma = numero1 + numero2; // suma números
20
21       // muestra los resultados en un diálogo de mensajes de JOptionPane
22       JOptionPane.showMessageDialog( null, "La suma es " + suma,
23          "Suma de dos enteros", JOptionPane.PLAIN_MESSAGE );
24    } // fin del método main
25 } // fin de la clase Suma
```





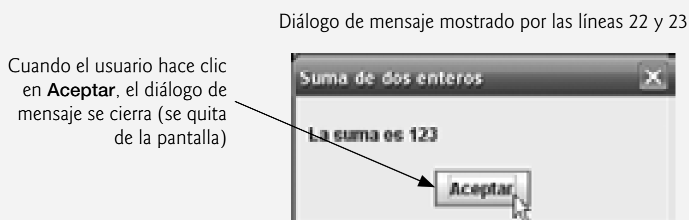


Figura 11.2 | Programa de suma que utiliza a `JOptionPane` para entrada y salida.

El usuario escribe caracteres en el campo de texto, y después hace clic en el botón **Aceptar** u oprime la tecla *Intro* para enviar el objeto `String` al programa. Al hacer clic en **Aceptar** también se **cierra** (**oculta**) **el diálogo**. [*Nota:* si escribe en el campo de texto y no aparece nada, actívelo haciendo clic sobre él con el ratón]. A diferencia de `Scanner`, que puede utilizarse para que el usuario introduzca valores de varios tipos mediante el teclado, un diálogo de entrada sólo puede introducir objetos `String`. Esto es común en la mayoría de los componentes de la GUI. Técnicamente, el usuario puede escribir cualquier cosa en el campo de texto del diálogo de entrada. Nuestro programa asume que el usuario introduce un valor entero válido. Si el usuario hace clic en el botón **Cancelar**, `showInputDialog` devuelve `null`. Si el usuario escribe un valor no entero o si hace clic en el botón **Cancelar** en el diálogo de entrada, se producirá un error lógico en tiempo de ejecución en este programa y no operará en forma correcta. El capítulo 13, Manejo de excepciones, habla acerca de cómo manejar dichos errores. Las líneas 12 y 13 muestran otro diálogo de entrada que pide al usuario que introduzca el segundo entero.

Convertir objetos `String` en valores `int`

Para realizar el cálculo en esta aplicación, debemos convertir los objetos `String` que el usuario introdujo, en valores `int`. En la sección 7.12 vimos que el método `static parseInt` de la clase `Integer` convierte su argumento `String` en un valor `int`. Las líneas 16 y 17 asignan los valores convertidos a las variables locales `numero1` y `numero2`. Después, la línea 19 suma estos valores y asigna el resultado a la variable local `suma`.

Diálogos de mensaje

Las líneas 22 y 23 usan el método `static` **`showMessageDialog`** de `JOptionPane` para mostrar un diálogo de mensaje (la última captura de pantalla de la figura 11.2) que contiene la suma. El primer argumento ayuda a la aplicación de Java a determinar en dónde debe colocar el cuadro de diálogo. El valor `null` indica que el diálogo debe aparecer en el centro de la pantalla de la computadora. El primer argumento puede usarse también para especificar que el diálogo debe aparecer centrado sobre una ventana específica, lo cual demostraremos más adelante en la sección 11.8. El segundo argumento es el mensaje a mostrar; en este caso, el resultado de concatenar el objeto `String` `"La suma es "` y el valor de `suma`. El tercer argumento (`"Suma de dos enteros"`) representa la cadena que debe aparecer en la barra de título del diálogo, en la parte superior. El cuarto argumento (**`JOptionPane.PLAIN_MESSAGE`**) es el tipo de diálogo de mensaje a mostrar. Un diálogo `PLAIN_MESSAGE` no muestra un icono a la izquierda del mensaje. La clase **`JOptionPane`** proporciona varias versiones sobrecargadas de los métodos `showInputDialog` y `showMessageDialog`, así como métodos que muestran otros tipos de diálogos. Para obtener información completa acerca de la clase `JOptionPane`, visite el sitio `java.sun.com/javase/6/docs/api/javax/swing/JOptionPane.html`.



Observación de apariencia visual 11.3

Por lo general, la barra de título de una ventana utiliza ***capitalización de título de libro****: un estilo que capitaliza la primera letra de cada palabra significativa en el texto, y no termina con ningún signo de puntuación (por ejemplo, Capitalización en el Título de un Libro).*

Constantes de diálogos de mensajes de `JOptionPane`

Las constantes que representan los tipos de diálogos de mensajes se muestran en la figura 11.3. Todos los tipos de diálogos de mensaje, excepto `PLAIN_MESSAGE`, muestran un icono a la izquierda del mensaje. Estos iconos proporcionan una indicación visual de la importancia del mensaje para el usuario. Observe que un icono `QUESTION_MESSAGE` es el icono predeterminado para un cuadro de diálogo de entrada (vea la figura 11.2).

Tipo de diálogo de mensaje	Icono	Descripción
`ERROR_MESSAGE`	(x)	Un diálogo que indica un error al usuario.
`INFORMATION_MESSAGE`	(i)	Un diálogo con un mensaje informativo para el usuario.

Figura 11.3 | Constantes `static` de `JOptionPane` para diálogos de mensaje. (Parte 1 de 2).

Tipo de diálogo de mensaje	Icono	Descripción
`WARNING_MESSAGE`	⚠	Un diálogo que advierte al usuario sobre un problema potencial.
`QUESTION_MESSAGE`	?	Un diálogo que hace una pregunta al usuario. Por lo general, este diálogo requiere una respuesta, como hacer clic en un botón **Sí** o **No**.
`PLAIN_MESSAGE`	sin icono	Un diálogo que contiene un mensaje, pero no un icono.

Figura 11.3 | Constantes `static` de `JOptionPane` para diálogos de mensaje. (Parte 2 de 2).

11.3 Generalidades de los componentes de Swing

Aunque es posible realizar operaciones de entrada y salida utilizando los diálogos de `JOptionPane` que presentamos en la sección 11.2, la mayoría de las aplicaciones de GUI requieren interfaces de usuario personalizadas y más elaboradas. El resto de este capítulo habla acerca de muchos componentes de la GUI que permiten a los desarrolladores de aplicaciones crear GUIs robustas. La figura 11.4 lista varios **componentes de la GUI de Swing** del paquete **`javax.swing`**, que se utilizan para crear GUIs en Java. La mayoría de los componentes de Swing son componentes **puros de Java**: están escritos, se manipulan y se muestran completamente en Java. Forman parte de las **JFC** (**Java Foundation Classes**); las bibliotecas de Java para el desarrollo de GUIs para múltiples plataformas. Visite `java.sun.com/products/jfc` para obtener más información acerca de JFC.

Comparación entre Swing y AWT

En realidad hay dos conjuntos de componentes de GUI en Javas. Antes de introducir a Swing en Java SE 1.2, las GUIs de Java se creaban a partir de componentes del **Abstract Window Toolkit** (**AWT**) en el paquete **`java.awt`**. Cuando una aplicación de Java con una GUI del AWT se ejecuta en distintas plataformas, los componentes de la GUI de la aplicación se muestran de manera distinta en cada plataforma. Considere una aplicación que muestra un objeto de tipo `Button` (paquete `java.awt`). En una computadora que ejecuta el sistema operativo Microsoft Windows, el objeto `Button` tendrá la misma apariencia que los botones en las demás aplicaciones Windows. De manera similar, en una computadora que ejecuta el sistema operativo Apple Mac OS X, el objeto `Button` tendrá la misma apariencia visual que los botones en las demás aplicaciones Macintosh. Algunas veces, la forma en la que un usuario puede interactuar con un componente específico del AWT difiere entre una plataforma y otra.

Componente	Descripción
`JLabel`	Muestra texto que no puede editarse, o iconos.
`JTextField`	Permite al usuario introducir datos mediante el teclado. También se puede utilizar para mostrar texto que puede o no editarse.
`JButton`	Activa un evento cuando se oprime mediante el ratón.
`JCheckBox`	Especifica una opción que puede seleccionarse o no seleccionarse.
`JComboBox`	Proporciona una lista desplegable de elementos, a partir de los cuales el usuario puede realizar una selección, haciendo clic en un elemento o posiblemente escribiendo en el cuadro.
`JList`	Proporciona una lista de elementos a partir de los cuales el usuario puede realizar una selección, haciendo clic en cualquier elemento en la lista. Pueden seleccionarse varios elementos.
`JPanel`	Proporciona un área en la que pueden colocarse y organizarse los componentes. También puede utilizarse como un área de dibujo para gráficos.

Figura 11.4 | Algunos componentes básicos de GUI.

En conjunto, a la apariencia y la forma en la que interactúa el usuario con la aplicación se les denomina la **apariencia visual**. Los componentes de GUI de Swing nos permiten especificar una apariencia visual uniforme para una aplicación a través de todas las plataformas, o para usar la apariencia visual personalizada de cada plataforma. Incluso, hasta una aplicación puede modificar la apariencia visual durante la ejecución, para permitir a los usuarios elegir su propia apariencia visual preferida.



Tip de portabilidad 11.1

Los componentes de Swing se implementan en Java, por lo que son más portables y flexibles que los componentes de GUI originales de Java del paquete `java.awt`*, que estaban basados en los componentes de GUI de la plataforma subyacente. Por esta razón, generalmente se prefieren los componentes de GUI de Swing.*

Comparación entre componentes de GUI ligeros y pesados

La mayoría de los componentes de Swing no están enlazados a los componentes reales de GUI que soporta la plataforma subyacente en la cual se ejecuta una aplicación. Dichos componentes de la GUI se conocen como **componentes ligeros**. Los componentes de AWT (muchos de los cuales se asemejan a los componentes de Swing) están enlazados a la plataforma local y se conocen como **componentes pesados**, ya que dependen del **sistema de ventanas** de la plataforma local para determinar su funcionalidad y su apariencia visual.

Varios componentes de Swing son componentes ligeros. Al igual que los componentes de AWT, los componentes de GUI pesados de Swing requieren interacción directa con el sistema de ventanas locales, el cual puede restringir su apariencia y funcionalidad, haciéndolos menos flexibles que los componentes ligeros.



Observación de apariencia visual 11.4

La apariencia visual de una GUI definida con componentes de GUI pesados del paquete `java.awt` *tal vez nunca varíe de una plataforma a otra. Debido a que los componentes pesados están enlazados a la GUI de la plataforma local, la apariencia visual varía de una plataforma a otra.*

Superclases de los componentes de GUI ligeros de Swing

El diagrama de clases de UML de la figura 11.5 muestra una jerarquía de herencia que contiene clases a partir de las cuales los componentes ligeros de Swing heredan sus atributos y comportamientos comunes. Como vimos en el capítulo 9, la clase `Object` es la superclase de la jerarquía de clases de Java.



Observación de ingeniería de software 11.1

Estudie los atributos y comportamientos de las clases en la jerarquía de clases de la figura 11.5. Estas clases declaran las características comunes para la mayoría de los componentes de Swing.

La clase **`Component`** (paquete `java.awt`) es una subclase de `Object` que declara muchos de los atributos y comportamientos comunes para los componentes de GUI en los paquetes `java.awt` y `javax.swing`. La mayoría

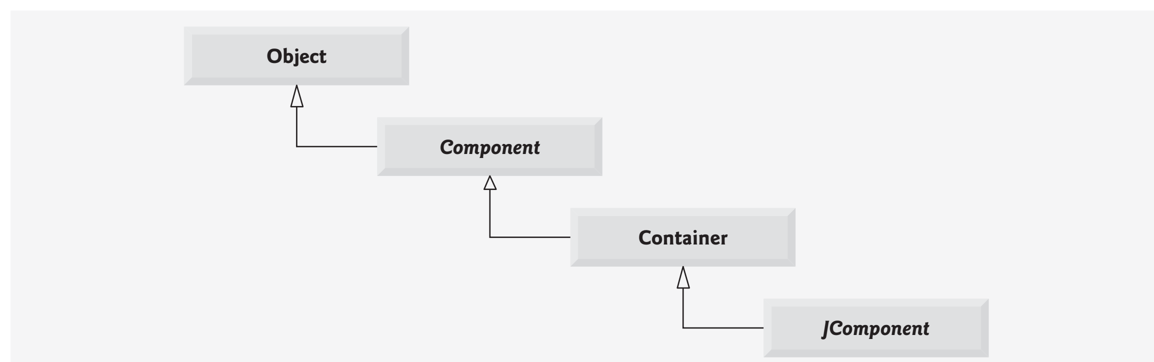


Figura 11.5 | Superclases comunes de muchos de los componentes de Swing.

de los componentes de GUI extienden la clase `Component` de manera directa o indirecta. Para obtener una lista completa de estas características comunes, visite `java.sun.com/javase/6/docs/api/java/awt/Component.html`.

La clase **`Container`** (paquete `java.awt`) es una subclase de `Component`. Como veremos pronto, los objetos `Component` se adjuntan a objetos `Container` (como las ventanas), de manera que los objetos `Component` se puedan organizar y mostrar en la pantalla. Cualquier objeto que *sea un* `Container` se puede utilizar para organizar a otros objetos `Component` en una GUI. Como un `Container` *es un* `Component`, puede adjuntar objetos `Container` a otros objetos `Container` para ayudar a organizar una GUI. Para obtener una lista completa de las características de `Container` que son comunes para los componentes ligeros de Swing, visite `java.sun.com/javase/6/docs/api/java/awt/Container.html`.

La clase **`JComponent`** (paquete `javax.swing`) es una subclase de `Container`. `JComponent` es la superclase de todos los componentes ligeros de Swing, y declara los atributos y comportamientos comunes. Debido a que `JComponent` es una subclase de `Container`, todos los componentes ligeros de Swing son también objetos `Container`. Algunas de las características comunes para los componentes ligeros que soporta `JComponent` son:

1. Una **apariencia visual adaptable**, la cual puede utilizarse para personalizar la apariencia de los componentes (por ejemplo, para usarlos en plataformas específicas). En la sección 22.6 veremos un ejemplo de esto.
2. Teclas de método abreviado (llamadas **nemónicos**) para un acceso directo a los componentes de la GUI por medio del teclado. En la sección 22.4 veremos un ejemplo de esto.
3. Herramientas manejadoras de eventos comunes, para casos en los que varios componentes de la GUI inician las mismas acciones en una aplicación.
4. Breves descripciones del propósito de un componente de la GUI (lo que se conoce como **cuadros de información sobre herramientas** o **tool tips**) que se muestran cuando el cursor del ratón se coloca sobre el componente durante un breve periodo. En la siguiente sección veremos un ejemplo de esto.
5. Soporte para tecnologías de ayuda, como lectores de pantalla Braille para las personas con impedimentos visuales.
6. Soporte para la **localización** de la interfaz de usuario; es decir, personalizar la interfaz de usuario para mostrarla en distintos lenguajes y utilizar las convenciones de la cultura local.

Éstas son sólo algunas de las muchas características de los componentes de Swing. Visite `java.sun.com/javase/6/docs/api/javax/swing/JComponent.html` para obtener más detalles de las características comunes de los componentes ligeros.

11.4 Mostrar texto e imágenes en una ventana

Nuestro siguiente ejemplo introduce un marco de trabajo para crear aplicaciones de GUI. Este marco de trabajo utiliza varios conceptos que verá en muchas de nuestras aplicaciones de GUI. Éste es nuestro primer ejemplo en el que la aplicación aparece en su propia ventana. La mayoría de las ventanas que creará son una instancia de la clase `JFrame` o una subclase de `JFrame`. `JFrame` proporciona los atributos y comportamientos básicos de una ventana: una barra de título en la parte superior, y botones para minimizar, maximizar y cerrar la ventana. Como la GUI de una aplicación por lo general es específica para esa aplicación, la mayoría de nuestros ejemplos consistirán en dos clases: una subclase de `JFrame` que nos ayuda a demostrar los nuevos conceptos de la GUI y una clase de aplicación, en la que `main` crea y muestra la ventana principal de la aplicación.

Etiquetado de componentes de la GUI

Una GUI típica consiste en muchos componentes. En una GUI extensa, puede ser difícil identificar el propósito de cada componente, a menos que el diseñador de la GUI proporcione instrucciones de texto o información que indique el propósito de cada componente. Dicho texto se conoce como **etiqueta** y se crea con la clase **`JLabel`**; una subclase de `JComponent`. Un objeto `JLabel` muestra una sola línea de texto de sólo lectura, una imagen, o texto y una imagen. Raras veces las aplicaciones modifican el contenido de una etiqueta, después de crearla.



Observación de apariencia visual 11.5

Por lo general, el texto en un objeto `JLabel` *utiliza la capitalización estilo oración.*

La aplicación de las figuras 11.6 y 11.7 demuestra varias características de `JLabel` y presenta el marco de trabajo que utilizamos en la mayoría de nuestros ejemplos de GUI. No resaltamos el código en este ejemplo, ya que casi todo es nuevo. [*Nota:* hay muchas más características para cada componente de GUI de las que podemos cubrir en nuestros ejemplos. Para conocer todos los detalles acerca de cada componente de la GUI, visite su página en la documentación en línea. Para la clase `JLabel`, visite `java.sun.com/javase/6/docs/api/javax/swing/JLabel.html`].

La clase `LabelFrame` (figura 11.6) es una subclase de `JFrame`. Utilizaremos una instancia de la clase `LabelFrame` para mostrar una ventana que contiene tres objetos `JLabel`. Las líneas 3 a 8 importan las clases utilizadas en la clase `LabelFrame`. La clase extiende a `JFrame` para heredar las características de una ventana. Las líneas 12 a 14 declaran las tres variables de instancia `JLabel`, cada una de las cuales se instancia en el constructor de `JLabelFrame` (líneas 17 a 41). Por lo general, el constructor de la subclase de `JFrame` crea la GUI que se muestra en la ventana, cuando se ejecuta la aplicación. La línea 19 invoca al constructor de la superclase `JFrame` con el argumento `"Prueba de JLabel"`. El constructor de `JFrame` utiliza este objeto `String` como el texto en la barra de título de la ventana.

```
1  // Fig. 11.6: LabelFrame.java
2  // Demostración de la clase JLabel.
3  import java.awt.FlowLayout; // especifica cómo se van a ordenar los componentes
4  import javax.swing.JFrame; // proporciona las características básicas de una ventana
5  import javax.swing.JLabel; // muestra texto e imágenes
6  import javax.swing.SwingConstants; // constantes comunes utilizadas con Swing
7  import javax.swing.Icon; // interfaz utilizada para manipular imágenes
8  import javax.swing.ImageIcon; // carga las imágenes
9
10 public class LabelFrame extends JFrame
11 {
12    private JLabel etiqueta1; // JLabel sólo con texto
13    private JLabel etiqueta2; // JLabel construida con texto y un icono
14    private JLabel etiqueta3; // JLabel con texto adicional e icono
15
16    // El constructor de LabelFrame agrega objetos JLabel a JFrame
17    public LabelFrame()
18    {
19       super( "Prueba de JLabel" );
20       setLayout( new FlowLayout() ); // establece el esquema del marco
21
22       // Constructor de JLabel con un argumento String
23       etiqueta1 = new JLabel( "Etiqueta con texto" );
24       etiqueta1.setToolTipText( "Esta es etiqueta1" );
25       add( etiqueta1 ); // agrega etiqueta1 a JFrame
26
27       // Constructor de JLabel con argumentos de cadena, Icono y alineación
28       Icon insecto = new ImageIcon( getClass().getResource( "insecto1.gif" ) );
29       etiqueta2 = new JLabel( "Etiqueta con texto e icono", insecto,
30          SwingConstants.LEFT );
31       etiqueta2.setToolTipText( "Esta es etiqueta2" );
32       add( etiqueta2 ); // agrega etiqueta2 a JFrame
33
34       etiqueta3 = new JLabel(); // Constructor de JLabel sin argumentos
35       etiqueta3.setText( "Etiqueta con icono y texto en la parte inferior" );
36       etiqueta3.setIcon( insecto ); // agrega icono a JLabel
37       etiqueta3.setHorizontalTextPosition( SwingConstants.CENTER );
```

Figura 11.6 | Objetos `JLabel` con texto e iconos. (Parte 1 de 2).

```
      etiqueta3.setVerticalTextPosition( SwingConstants.BOTTOM );
      etiqueta3.setToolTipText( "Esta es etiqueta3" );
      add( etiqueta3 ); // agrega etiqueta3 a JFrame
   } // fin del constructor de LabelFrame
} // fin de la clase LabelFrame
```

Figura 11.6 | Objetos `JLabel` con texto e iconos. (Parte 2 de 2).

```
// Fig. 11.7: PruebaLabel.java
// Prueba de LabelFrame.
import javax.swing.JFrame;

public class PruebaLabel
{
   public static void main( String args[] )
   {
      LabelFrame marcoEtiqueta = new LabelFrame(); // crea objeto LabelFrame
      marcoEtiqueta.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
      marcoEtiqueta.setSize( 275, 180 ); // establece el tamaño del marco
      marcoEtiqueta.setVisible( true ); // muestra el marco
   } // fin de main
} // fin de la clase PruebaLabel
```



Figura 11.7 | Clase de prueba de `LabelFrame`.

Especificación del esquema

Al crear una GUI, cada componente de ésta debe adjuntarse a un contenedor, como una ventana creada con un objeto `JFrame`. Además, por lo general debemos decidir en dónde colocar cada componente de la GUI. Esto se conoce como especificar el esquema de los componentes de la GUI. Como aprenderá al final de este capítulo y en el capítulo 22, Componentes de la GUI: parte 2, Java cuenta con varios **administradores de esquemas** que pueden ayudarle a colocar los componentes.

Muchos entornos de desarrollo integrados (IDE) proporcionan herramientas de diseño de GUIs, en las cuales podemos especificar el tamaño y la ubicación exactos de un componente en forma visual utilizando el ratón, y después el IDE genera el código de la GUI por nosotros. Aunque dichos IDEs pueden simplificar considerablemente la creación de GUIs, sus capacidades son distintas.

Para asegurar que el código en este libro pueda utilizarse con cualquier IDE, no utilizamos un IDE para crear el código de la GUI en la mayoría de nuestros ejemplos. Por esta razón, usamos administradores de esquemas de Java en nuestros ejemplos de GUI. Uno de esos administradores es **`FlowLayout`**, en el cual los componentes de la GUI se colocan en un contenedor de izquierda a derecha, en el orden en el que el programa los une al contenedor. Cuando no hay más espacio para acomodar los componentes de izquierda a derecha, se siguen mostrando de izquierda a derecha en la siguiente línea. Si se cambia el tamaño del contenedor, un esquema `FlowLayout` reordena los componentes para dar cabida a la nueva anchura del contenedor, posiblemente con menos o más filas de componentes de la GUI. La línea 20 especifica que el esquema del objeto `LabelFrame` debe ser `FlowLayout`.

El método **`setLayout`** se hereda en la clase `LabelFrame`, indirectamente de la clase `Container`. El argumento para el método debe ser un objeto de una clase que implemente la interfaz `LayoutManager` (es decir, `FlowLayout`). La línea 20 crea un nuevo objeto `FlowLayout` y pasa su referencia como argumento para `setLayout`.

Cómo crear y adjuntar `etiqueta1`

Ahora que hemos especificado el esquema de la ventana, podemos empezar a crear y adjuntar componentes de la GUI en la ventana. La línea 23 crea un objeto `JLabel` y pasa `"Etiqueta con texto"` al constructor. El objeto `JLabel` muestra este texto en la pantalla como parte de la GUI de la aplicación. La línea 24 utiliza el método **`setToolTipText`** (heredado por `JLabel` de `JComponent`) para especificar la información sobre herramientas que se muestra cuando el usuario coloca el cursor del ratón sobre el objeto `JLabel` en la GUI. En la segunda captura de pantalla de la figura 11.7 puede ver un cuadro de información sobre herramientas de ejemplo. Cuando ejecute esta aplicación, trate de colocar el ratón sobre cada objeto `JLabel` para ver su información sobre herramientas. La línea 25 adjunta `etiqueta1` al objeto `LabelFrame`, para lo cual pasa `etiqueta1` al método **`add`**, que se hereda indirectamente de la clase `Container`.



Error común de programación 11.1

Si no agrega explícitamente un componente de GUI a un contenedor, el componente no se mostrará cuando aparezca el contenedor en la pantalla.



Observación de apariencia visual 11.6

Use cuadros de información sobre herramientas para agregar texto descriptivo a sus componentes de GUI. Este texto ayuda al usuario a determinar el propósito del componente de GUI en la interfaz de usuario.

Cómo crear y adjuntar `etiqueta2`

Los iconos son una forma popular de mejorar la apariencia visual de una aplicación, y también se utilizan comúnmente para indicar funcionalidad. Por ejemplo, la mayoría de las VCRs y los reproductores de DVD de la actualidad utilizan el mismo icono para reproducir una cinta o un DVD. Varios componentes de Swing pueden mostrar imágenes. Por lo general, un icono se especifica con un argumento **`Icon`** para un constructor o para el método **`setIcon`** del componente. Un `Icon` es un objeto de cualquier clase que implemente a la interfaz `Icon` (paquete `javax.swing`). Una de esas clases es **`ImageIcon`** (paquete `javax.swing`), que soporta varios formatos de imágenes, incluyendo: **(GIF) Formato de intercambio de gráficos**, **(PNG) Gráficos portables de red** y **(JPEG) Grupo de expertos unidos en fotografía**. Los nombres de archivos para cada uno de estos tipos termina con `.gif`, `.png` o `.jpg` (o `.jpeg`), respectivamente. En el capítulo 21, Multimedia: applets y aplicaciones, hablaremos sobre las imágenes con más detalle.

La línea 28 declara un objeto `ImageIcon`. El archivo `insecto1.gif` contiene la imagen a cargar y almacenar en el objeto `ImageIcon`. (Esta imagen se incluye en el directorio para este ejemplo, en el CD que acompaña a este libro). El objeto `ImageIcon` se asigna a la referencia `Icon` llamada `insecto`. Recuerde que la clase `ImageIcon` implementa a la interfaz `Icon`; un objeto `ImageIcon` *es un* objeto `Icon`.

En la línea 28, la expresión `getClass().getResource( "insecto1.gif" )` invoca al método **`getClass`** (heredado de la clase `Object`) para obtener una referencia al objeto `Class` que representa la declaración de la clase `LabelFrame`. Después, esa referencia se utiliza para invocar al método **`getResource`** de `Class`, el cual devuelve la ubicación de la imagen como un URL. El constructor de `ImageIcon` utiliza el URL para localizar la imagen, y después la carga en la memoria. Como vimos en el capítulo 1, la JVM carga las declaraciones de las clases en la memoria, usando un cargador de clases. El cargador de clases sabe en dónde se encuentra localizada en el disco cada clases que carga. El método `getResource` utiliza el cargador de clases del objeto `Class` para determinar la ubicación de un recurso, como un archivo de imagen. En este ejemplo, el archivo de imagen se almacena en la misma ubicación que el archivo `LabelFrame.class`. Las técnicas aquí descritas permiten que una aplicación cargue archivos de imagen de ubicaciones que son relativas al archivo `.class` de `LabelFrame` en disco.

Un objeto `JLabel` puede mostrar un objeto `Icon`. Las líneas 29 y 30 utilizan otro constructor de `JLabel` para crear un objeto `JLabel` que muestre el texto `"Etiqueta con texto e icono"` y el objeto `Icon` llamado `insecto` que se creó en la línea 28. El último argumento del constructor indica que el contenido de la etiqueta está justificado a la izquierda, o **alineado a la izquierda** (es decir, el icono y el texto se encuentran en el lado

izquierdo del área de la etiqueta en la pantalla). La interfaz **SwingConstants** (paquete javax.swing) declara un conjunto de constantes enteras comunes (como SwingConstants.LEFT) que se utilizan con muchos componentes de Swing. De manera predeterminada, el texto aparece a la derecha de una imagen cuando una etiqueta contiene tanto texto como una imagen. Observe que las alineaciones horizontal y vertical de un objeto JLabel se pueden establecer mediante los métodos **setHorizontalAlignment** y **setVerticalAlignment**, respectivamente. La línea 31 especifica el texto de información sobre herramientas para etiqueta2, y la línea 32 agrega etiqueta2 al objeto JFrame.

Cómo crear y adjuntar etiqueta3

La clase JLabel cuenta con muchos métodos para modificar la apariencia de una etiqueta, una ves que se crea una instancia de ésta. La línea 34 crea un objeto JLabel e invoca a su constructor sin argumentos. Al principio, dicha etiqueta no tiene texto ni objeto Icon. La línea 35 utiliza el método **setText** de JLabel para establecer el texto mostrado en la etiqueta. El método correspondiente **getText** obtiene el texto actual mostrado en la etiqueta. La línea 36 utiliza el método **setIcon** de JLabel para especificar el objeto Icon a mostrar en la etiqueta. El correspondiente método **getIcon** obtiene el objeto Icon actual mostrado en una etiqueta. Las líneas 37 y 38 utilizan los métodos **setHorizontalTextPosition** y **setVerticalTextPosition** de JLabel para especificar la siguiente posición del texto en la etiqueta. En este caso, el texto se centrará en forma horizontal y aparecerá en la parte inferior de la etiqueta. Por ende, el objeto Icon aparecerá por encima del texto. Las constantes de posición horizontal en SwingConstants son LEFT, CENTER y RIGHT (figura 11.8). Las constantes de posición vertical en SwingConstants son TOP, CENTER y BOTTOM (figura 11.8). La línea 39 establece el texto de información sobre herramientas para etiqueta3. La línea 40 agrega etiqueta3 al objeto JFrame.

Constante	Descripción
Constantes de posición horizontal	
SwingConstants.LEFT	Coloca el texto a la izquierda.
SwingConstants.CENTER	Coloca el texto en el centro.
SwingConstants.RIGHT	Coloca el texto a la derecha.
Constantes de posición vertical	
SwingConstants.TOP	Coloca el texto en la parte superior.
SwingConstants.CENTER	Coloca el texto en el centro.
SwingConstants.BOTTOM	Coloca el texto en la parte inferior.

Figura 11.8 | Algunos componentes de GUI básicos.

Cómo crear y mostrar una ventana LabelFrame

La clase PruebaLabel (figura 11.7) crea un objeto de la clase LabelFrame (línea 9) y después especifica la operación de cierre predeterminada para la ventana. De manera predeterminada, al cerrar una ventana ésta simplemente se oculta. Sin embargo, cuando el usuario cierre la ventana LabelFrame, nos gustaría que la aplicación terminara. La línea 10 invoca al método **setDefaultCloseOperation** de LabelFrame (heredado de la clase JFrame) con la constante **JFrame.EXIT_ON_CLOSE** como el argumento para indicar que el programa debe terminar cuando el usuario cierre la ventana. Esta línea es importante. Sin ella, la aplicación no terminará cuando el usuario cierre la ventana. A continuación, la línea 11 invoca el método **setSize** de LabelFrame para especificar la anchura y la altura de la ventana. Por último, la línea 12 invoca al método **setVisible** de LabelFrame con el argumento true, para mostrar la ventana en la pantalla. Pruebe a cambiar el tamaño de la ventana, para ver cómo el esquema FlowLayout cambia las posiciones de los objetos JLabel, a medida que cambia la anchura de la ventana.

11.5 Campos de texto y una introducción al manejo de eventos con clases anidadas

Por lo general, un usuario interactúa con la GUI de una aplicación para indicar las tareas que ésta debe realizar. Por ejemplo, cuando usted escribe un mensaje en una aplicación de correo electrónico, al hacer clic en el botón **Enviar** le indica a la aplicación que envíe el correo electrónico a las direcciones especificadas. Las GUIs son **controladas por eventos**. Cuando el usuario interactúa con un componente de la GUI, la interacción (conocida como un **evento**) controla el programa para que realice una tarea. Algunos eventos (interacciones del usuario) comunes que podrían hacer que una aplicación realizara una tarea incluyen el hacer clic en un botón, escribir en un campo de texto, seleccionar un elemento de un menú, cerrar una ventana y mover el ratón. El código que realiza una tarea en respuesta a un evento se llama **manejador de eventos** y al proceso en general de responder a los eventos se le conoce como **manejo de eventos**.

En esta sección, presentaremos dos nuevos componentes de GUI que pueden generar eventos: **`JTextField`** y **`JPasswordField`** (paquete `javax.swing`).La clase `JTextField` extiende a la clase **`JTextComponent`** (paquete `javax.swing.text`), que proporciona muchas características comunes para los componentes de Swing basados en texto. La clase `JPasswordField` extiende a `JTextField` y agrega varios métodos específicos para el procesamiento de contraseñas. Cada uno de estos componentes es un área de una sola línea, en la cual el usuario puede introducir texto mediante el teclado. Las aplicaciones también pueden mostrar texto en un objeto `JTextField` (vea la salida de la figura 11.10). Un objeto `JPasswordField` muestra que se están escribiendo caracteres a medida que el usuario los introduce, pero oculta los caracteres reales con un **carácter de eco**, asumiendo que representan una contraseña que sólo el usuario debe conocer.

Cuando el usuario escribe datos en un objeto `JTextField` o `JPasswordField` y después oprime *Intro*, ocurre un evento. Nuestro siguiente ejemplo demuestra cómo un programa puede realizar una tarea en respuesta a ese evento. Las técnicas que se muestran aquí se pueden aplicar a todos los componentes de GUI que generen eventos.

La aplicación de las figuras 11.9 y 11.10 utiliza las clases `JTextField` y `JPasswordField` para crear y manipular cuatro campos de texto. Cuando el usuario escribe en uno de los campos de texto y después oprime *Intro*, la aplicación muestra un cuadro de diálogo de mensaje que contiene el texto que escribió el usuario. Sólo podemos escribir en el campo de texto que esté "**enfocado**". Un componente recibe el enfoque cuando el usuario hace clic en ese componente. Esto es importante, ya que el campo de texto con el enfoque es el que genera un evento cuando el usuario oprime *Intro*. En este ejemplo, cuando el usuario oprime *Intro* en el objeto `JPasswordField`, se revela la contraseña. Empezaremos por hablar sobre la preparación de la GUI, y después sobre el código para manejar eventos.

```
// Fig. 11.9: CampoTextoMarco.java
// Demostración de la clase JTextField.
import java.awt.FlowLayout;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import javax.swing.JFrame;
import javax.swing.JTextField;
import javax.swing.JPasswordField;
import javax.swing.JOptionPane;

public class CampoTextoMarco extends JFrame
{
   private JTextField campoTexto1; // campo de texto con tamaño fijo
   private JTextField campoTexto2; // campo de texto construido con texto
   private JTextField campoTexto3; // campo de texto con texto y tamaño
   private JPasswordField campoContrasenia; // campo de contraseña con texto

   // El constructor de CampoTextoMarco agrega objetos JTextField a JFrame
   public CampoTextoMarco()
```

Figura 11.9 | Objetos `JTextField` y `JPasswordField`. (Parte 1 de 3).

```
   {
      super( "Prueba de JTextField y JPasswordField" );
      setLayout( new FlowLayout() ); // establece el esquema del marco

      // construye campo de texto con 10 columnas
      campoTexto1 = new JTextField( 10 );
      add( campoTexto1 ); // agrega campoTexto1 a JFrame

      // construye campo de texto con texto predeterminado
      campoTexto2 = new JTextField( "Escriba el texto aqui" );
      add( campoTexto2 ); // agrega campoTexto2 a JFrame

      // construye campo de texto con texto predeterminado y 21 columnas
      campoTexto3 = new JTextField( "Campo de texto no editable", 21 );
      campoTexto3.setEditable( false ); // deshabilita la edición
      add( campoTexto3 ); // agrega campoTexto3 a JFrame

      // construye campo de contraseña con texto predeterminado38
      campoContrasenia = new JPasswordField( "Texto oculto" );
      add( campoContrasenia ); // agrega campoContrasenia a JFrame

      // registra los manejadores de eventos
      ManejadorCampoTexto manejador = new ManejadorCampoTexto();
      campoTexto1.addActionListener( manejador );
      campoTexto2.addActionListener( manejador );
      campoTexto3.addActionListener( manejador );
      campoContrasenia.addActionListener( manejador );
   } // fin del constructor de CampoTextoMarco

   // clase interna privada para el manejo de eventos
   private class ManejadorCampoTexto implements ActionListener
   {
      // procesa los eventos de campo de texto
      public void actionPerformed( ActionEvent evento )
      {
         String cadena = ""; // declara la cadena a mostrar

         // el usuario oprimió Intro en el objeto JTextField campoTexto1
         if ( evento.getSource() == campoTexto1 )
            cadena = String.format( "campoTexto1: %s",
               evento.getActionCommand() );

         // el usuario oprimió Intro en el objeto JTextField campoTexto2
         else if ( evento.getSource() == campoTexto2 )
            cadena = String.format( "campoTexto2: %s",
               evento.getActionCommand() );

         // el usuario oprimió Intro en el objeto JTextField campoTexto3
         else if ( evento.getSource() == campoTexto3 )
            cadena = String.format( "campoTexto3: %s",
               evento.getActionCommand() );

         // el usuario oprimió Intro en el objeto JTextField campoContrasenia
         else if ( evento.getSource() == campoContrasenia )
            cadena = String.format( "campoContrasenia: %s",
               new String( campoContrasenia.getPassword() ) );

         // muestra el contenido del objeto JTextField
         JOptionPane.showMessageDialog( null, cadena );
```

Figura 11.9 | Objetos `JTextField` y `JPasswordField`. (Parte 2 de 3).

```
79          } // fin del método actionPerformed
80       } // fin de la clase interna privada ManejadorCampoTexto
81    } // fin de la clase CampoTextoMarco
```

Figura 11.9 | Objetos `JTextField` y `JPasswordField`. (Parte 3 de 3).

Las líneas 3 a 9 importan las clases e interfaces que utilizamos en este ejemplo. La clase `CampoTextoMarco` extiende a `JFrame` y declara tres variables `JTextField` y una variable `JPasswordField` (líneas 13 a 16). Cada uno de los correspondientes campos de texto se instancia y se adjunta al objeto `CampoTextoMarco` en el constructor (líneas 19 a 47).

```
// Fig. 11.10: PruebaCampoTexto.java
// Prueba de CampoTextoMarco.
import javax.swing.JFrame;

public class PruebaCampoTexto
{
   public static void main( String args[] )
   {
      CampoTextoMarco campoTextoMarco = new CampoTextoMarco();
      campoTextoMarco.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
      campoTextoMarco.setSize( 350, 100 ); // establece el tamaño del marco
      campoTextoMarco.setVisible( true ); // muestra el marco
   } // fin de main
} // fin de la clase PruebaCampoTexto
```

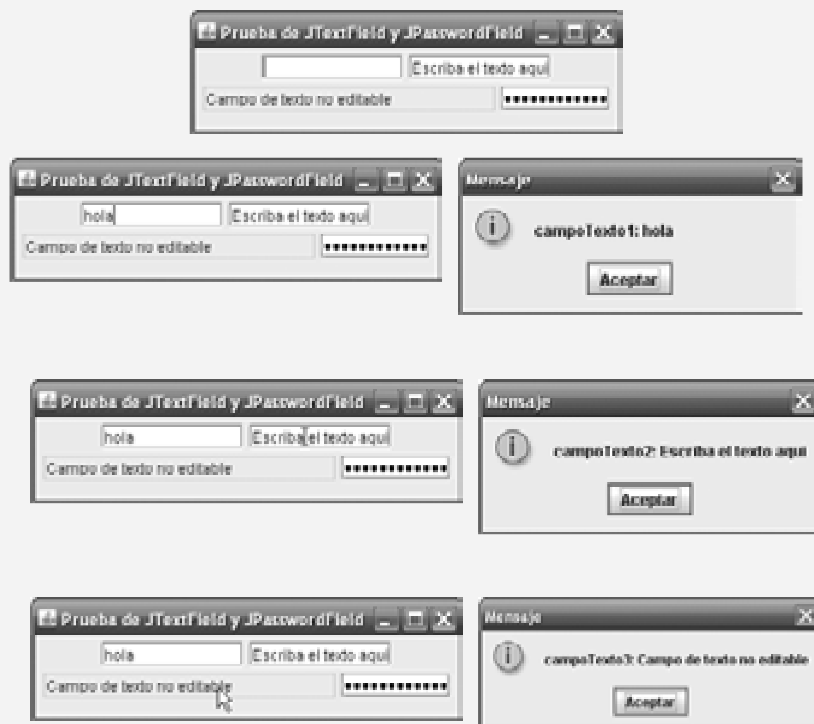


Figura 11.10 | Clase de prueba de `CampoTextoMarco`. (Parte 1 de 2).



Figura 11.10 | Clase de prueba de `CampoTextoMarco`. (Parte 2 de 2).

Creación de la GUI

La línea 22 establece el esquema del objeto `CampoTextoMarco` a `FlowLayout`. La línea 25 crea el objeto `campoTexto1` con 10 columnas de texto. La anchura en píxeles de una columna de texto se determina en base a la anchura promedio de un carácter en el tipo de letra actual del campo de texto. Cuando se muestra texto en un campo de texto, y el texto es más ancho que el campo de texto en sí, no está visible una parte del texto del lado derecho. Si usted escribe en un campo de texto y el cursor llega al extremo derecho del campo, el texto en el extremo izquierdo se empuja hacia el lado izquierdo del campo de texto y ya no estará visible. Los usuarios pueden usar las flechas de dirección izquierda y derecha para recorrer el texto completo, aun cuando éste no se pueda ver todo a la vez. La línea 26 agrega el objeto `campoTexto1` al objeto `JFrame`.

La línea 29 crea el objeto `campoTexto2` con el texto inicial `"Escriba el texto aqui"` para mostrarlo en el campo de texto. La anchura del campo se determina en base al texto predeterminado especificado en el constructor. La línea 30 agrega el objeto `campoTexto2` al objeto `JFrame`.

La línea 33 crea el objeto `campoTexto3` y llama al constructor de `JTextField` con dos argumentos: el texto predeterminado `"Campo de texto no editable"` para mostrarlo y el número de columnas (`21`). La anchura del campo de texto se determina en base al número de columnas especificadas. La línea 34 utiliza el método **`setEditable`** (heredado por `JTextField` de la clase `JTextComponent`) para hacer el campo de texto no editable; es decir, el usuario no puede modificar el texto. La línea 35 agrega el objeto `campoTexto3` al objeto `JFrame`.

La línea 38 crea `campoContrasenia` con el texto `"Texto oculto"` a mostrar en el campo de texto. La anchura de este campo de texto se determina en base a la anchura del texto predeterminado. Al ejecutar la aplicación, observe que el texto se muestra como una cadena de asteriscos. La línea 39 agrega `campoContrasenia` al objeto `JFrame`.

Pasos requeridos para establecer el manejo de eventos para un componente de GUI

Este ejemplo debe mostrar un diálogo de mensaje que contenga el texto de un campo de texto, cuando el usuario oprime *Intro* en ese campo de texto. Antes de que una aplicación pueda responder a un evento para un componente de GUI específico, debemos realizar varios pasos de codificación:

1. Crear una clase que represente al manejador de eventos.
2. Implementar una interfaz apropiada, conocida como **interfaz de escucha de eventos**, en la clase del *paso 1*.
3. Indicar que se debe notificar a un objeto de la clase de los *pasos 1* y *2* cuando ocurra el evento. A esto se le conoce como **registrar el manejador de eventos**.

Uso de una clase anidada para implementar un manejador de eventos

Todas las clases que hemos visto hasta ahora se conocen como **clases de nivel superior**; es decir, las clases no se declararon dentro de otras clases. Java nos permite declarar clases dentro de otras clases; a éstas se les conoce como **clases anidadas**. Las clases anidadas pueden ser `static` o no `static`. Las clases anidadas no `static` se llaman **clases internas**, y se utilizan con frecuencia para el manejo de eventos.



Observación de ingeniería de software 11.2

Una clase interna puede acceder directamente a las variables y métodos de su clase de nivel superior, aun cuando sean `private`.

Antes de poder crear un objeto de una clase interna, debe haber primero un objeto de la clase de nivel superior que contenga a la clase interna. Esto se requiere debido a que un objeto de la clase interna tiene implícitamente una referencia a un objeto de su clase de nivel superior. También hay una relación especial entre estos objetos: el objeto de la clase interna puede acceder directamente a todas las variables de instancia y métodos de la clase externa. Una clase interna que es `static` no requiere un objeto de su clase de nivel superior, y no tiene implícitamente una referencia a un objeto de la clase de nivel superior. Como veremos en el capítulo 12, Gráficos y Java 2D™, la API 2D de Java utiliza mucho las clases anidadas `static`.

El manejo de eventos en este ejemplo se realiza mediante un objeto de la clase interna `private ManejadorCampoTexto` (líneas 50 a 80). Esta clase es `private` debido a que se utilizará sólo para crear manejadores de eventos para los campos de texto en la clase de nivel superior `CampoTextoMarco`. Al igual que con los otros miembros de una clase, las clases internas pueden declararse como `public`, `protected` o `private`.

Los componentes de GUI pueden generar una variedad de eventos en respuesta a las interacciones del usuario. Cada evento se representa mediante una clase, y sólo puede procesarse mediante el tipo apropiado de manejador de eventos. En la mayoría de los casos, los eventos que soporta un componente de GUI se describen en la documentación de la API de java para la clase de ese componente y sus superclases. Cuando el usuario oprime *Intro* en un objeto `JTextField` o `JPasswordField`, el componente de GUI genera un evento **`ActionEvent`** (paquete `java.awt.event`). Dicho evento se procesa mediante un objeto que implementa la interfaz **`ActionListener`** (paquete `java.awt.event`). La información aquí descrita está disponible en la documentación de la API de Java para las clases `JTextField` y `ActionEvent`. Como `JPasswordField` es una subclase de `JTextField`, `JPasswordField` soporta los mismos eventos.

Para prepararnos para manejar los eventos en este ejemplo, la clase interna `ManejadorCampoTexto` implementa la interfaz `ActionListener` y declara el único método en esa interfaz: `actionPerformed` (líneas 53 a 79). Este método especifica las tareas a realizar cuando ocurre un evento `ActionEvent`. Por lo tanto, la clase `TextFieldHandler` cumple con los *pasos 1* y *2* que se listaron anteriormente en esta sección. En breve hablaremos sobre los detalles del método `actionPerformed`.

Registro del manejador de evento para cada campo de texto

En el constructor de `MarcoCampoTexto`, la línea 42 crea un objeto `ManejadorCampoTexto` y lo asigna a la variable `manejador`. El método `actionPerformed` de este objeto se llamará en forma automática cuando el usuario oprima *Intro* en cualquiera de los campos de texto de la GUI. Sin embargo, antes de que pueda ocurrir esto, el programa debe registrar este objeto como el manejador de eventos para cada campo de texto. Las líneas 43 a 46 son las instrucciones de registro de eventos que especifican a `manejador` como el manejador de eventos para los tres objetos `JTextField` y el objeto `JPasswordField`. La aplicación llama al método **`addActionListener`** de `JTextField` para registrar el manejador de eventos para cada componente. Este método recibe como argumento un objeto `ActionListener`, el cual puede ser un objeto de cualquier clase que implemente a `ActionListener`. El objeto `manejador` *es un* `ActionListener`, ya que la clase `ManejadorCampoTexto` implementa a `ActionListener`. Una vez que se ejecutan las líneas 43 a 46, el objeto **manejador escucha los eventos**. Ahora, cuando el usuario oprime *Intro* en cualquiera de estos cuatro campos de texto, se hace una llamada al método `actionPerformed` (líneas 53 a 79) en la clase `ManejadorCampoTexto` para que maneje el evento. Si no está registrado un manejador de eventos para un campo de texto específico, el evento que ocurre cuando el usuario oprime *Intro* en ese campo se **consume** (es decir, la aplicación simplemente lo ignora).



Observación de ingeniería de software 11.3

El componente de escucha de eventos para cierto evento debe implementar a la interfaz de escucha de eventos apropiada.



Error común de programación 11.2

Olvidar registrar un objeto manejador de eventos para un tipo de evento específico de un componente de la GUI hace que los eventos de ese tipo se ignoren.

Detalles del método `actionPerformed` *de la clase* `ManejadorCampoTexto`

En este ejemplo estamos usando el método `actionPerformed` de un objeto manejador de eventos (líneas 53 a 79) para manejar los eventos generados por cuatro campos de texto. Como nos gustaría imprimir en pantalla el

nombre de la variable de instancia de cada campo de texto para fines demostrativos, debemos determinar cuál campo de texto generó el evento cada vez que se hace una llamada a `actionPerformed`. El componente de GUI con el que interactúa el usuario es el **origen del evento**. En este ejemplo, el origen del evento es uno de los cuatro campos de texto o el campo de contraseña. Cuando el usuario oprime *Intro* mientras uno de estos componentes de GUI tiene el enfoque, el sistema crea un objeto `ActionEvent` único que contiene información acerca del evento que acaba de ocurrir, como el origen del evento y el texto en el campo de texto. Después, el sistema pasa este objeto `ActionEvent` en una llamada al método `actionPerformed` del componente de escucha de eventos. En este ejemplo, mostramos parte de esa información en un diálogo de mensaje. La línea 55 declara el objeto `String` que se va a mostrar. La variable se inicializa con la **cadena vacía**; una cadena que no contiene caracteres. El compilador requiere esto, en caso de que no se ejecute ninguna de las bifurcaciones de la instrucción `if` anidada en las líneas 58 a 75.

El método `getSource` de `ActionEvent` (que se llama en las líneas 58, 63, 68 y 73) devuelve una referencia al origen del evento. La condición en la línea 58 pregunta, "¿Es `campoTexto1` el **origen del evento**? Esta condición compara las referencias en ambos lados del operador `==` para determinar si se refieren al mismo objeto. Si ambos se refieren a `campoTexto1`, entonces el programa sabe que el usuario oprimió *Intro* en `campoTexto1`. En este caso, las líneas 59 y 60 crean un objeto `String` que contiene el mensaje que la línea 78 mostrará en un diálogo de mensaje. La línea 60 utiliza el método **`getActionCommand`** de `ActionEvent` para obtener el texto que escribió el usuario en el campo de texto que generó el evento.

Si el usuario interactuó con el objeto `JPasswordField`, las líneas 74 y 75 utilizan el método **`getPassword`** de `JPasswordField` para obtener la contraseña y crear el objeto `String` a mostrar. Este método devuelve la contraseña como un arreglo de tipo `char`, que se utiliza como argumento para un constructor de `String`, para crear una cadena que contenga los caracteres en el arreglo.

La clase **`PruebaCampoTexto`**

La clase `PruebaCampoTexto` (figura 11.10) contiene el método `main` que ejecuta esta aplicación y muestra un objeto de la clase `CampoTextoMarco`. Al ejecutar la aplicación, observe que hasta el campo `JTextField` (`campoTexto3`) puede generar un evento `ActionEvent`. Para probar esto, haga clic en el campo de texto para darle el enfoque y después oprima *Intro*. Observe además que el texto actual de la contraseña se muestra al oprimir *Intro* en el campo `JPasswordField`. ¡Desde luego que, por lo general, no se debe mostrar la contraseña!

Esta aplicación usó un solo objeto de la clase `ManejadorCampoTexto` como el componente de escucha de eventos para cuatro campos de texto. Empezando en la sección 11.9, verá que es posible declarar varios objetos de escucha de eventos del mismo tipo, y registrar cada objeto individual para cada evento de un componente de la GUI. Esta técnica nos permite eliminar la lógica `if...else` utilizada en el manejador de eventos de este ejemplo, al proporcionar manejadores de eventos separados para los eventos de cada componente.

11.6 Tipos de eventos comunes de la GUI e interfaces de escucha

En la sección 11.5 aprendió que la información acerca del evento que ocurre cuando el usuario oprime *Intro* en un campo de texto se almacena en un objeto `ActionEvent`. Pueden ocurrir muchos tipos distintos de eventos cuando el usuario interactúa con una GUI. La información acerca de cualquier evento de GUI que ocurre se almacena en un objeto de una clase que extiende a `AWTEvent`. La figura 11.11 ilustra una jerarquía que contiene muchas clases de eventos del paquete **`java.awt.event`**. Algunas de éstas se describen en este capítulo y en el capítulo 22. Estos tipos de eventos se utilizan tanto con componentes de AWT como de Swing. Los tipos de eventos adicionales que son específicos para los componentes de GUI de Swing se declaran en el paquete **`javax.swing.event`**.

Resumiremos las tres partes requeridas para el mecanismo de manejo de eventos que vimos en la sección 11.5: el origen del evento, el objeto del evento y el componente de escucha del evento. El origen del evento es el componente específico de la GUI con el que interactúa el usuario. El objeto del evento encapsula información acerca del evento que ocurrió, como una referencia al origen del evento, y cualquier información específica del evento que pueda requerir el componente de escucha del evento, para que pueda manejarlo. El componente de escucha del evento es un objeto que recibe una notificación del origen del evento cuando éste ocurre; en efecto, "escucha" un evento, y uno de sus métodos se ejecuta en respuesta al evento. Un método del componente de escucha del evento recibe un objeto evento cuando se notifica al componente de escucha acerca del evento. Después,

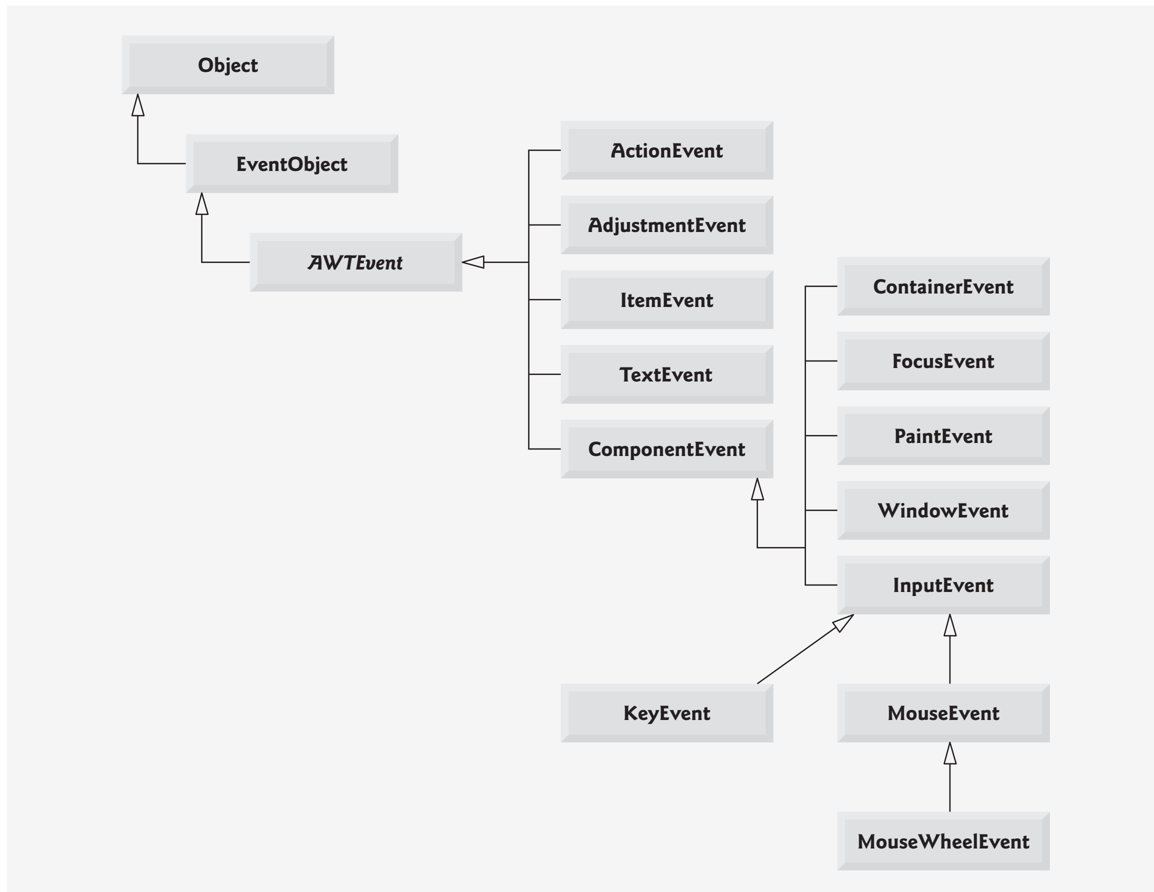


Figura 11.11 | Algunas clases de eventos del paquete `java.awt.event`.

el componente de escucha del evento utiliza el objeto evento para responder al evento. El modelo de manejo de eventos que se describe aquí se conoce como **modelo de eventos por delegación**: el procesamiento de un evento se delega a un objeto específico (el componente de escucha de eventos) en la aplicación.

Para cada tipo de objeto evento hay, por lo general, una interfaz de escucha de eventos que le corresponde. Un componente de escucha de eventos para un evento de GUI es un objeto de una clase que implementa a una o más de las interfaces de escucha de eventos de los paquetes `java.awt.event` y `javax.swing.event`. Muchos de los tipos de componentes de escucha de eventos son comunes para los componentes de Swing y de AWT. Dichos tipos se declaran en el paquete `java.awt.event`, y algunos de ellos se muestran en la figura 11.12. Los tipos de escucha de eventos adicionales, específicos para los componentes de Swing, se declaran en el paquete `javax.swing.event`.

Cada interfaz de escucha de eventos especifica uno o más métodos manejadores de eventos que deben declararse en la clase que implementa a la interfaz. En la sección 10.7 vimos que cualquier clase que implementa a una interfaz debe declarar a todos los métodos `abstract` de esa interfaz; en caso contrario, la clase es `abstract` y no puede utilizarse para crear objetos.

Cuando ocurre un evento, el componente de la GUI con el que el usuario interactuó notifica a sus componentes de escucha registrados, llamando al método de manejo de eventos apropiado de cada componente de escucha. Por ejemplo, cuando el usuario oprime la tecla *Intro* en un objeto `JTextField`, se hace una llamada al método `actionPerformed` del componente de escucha registrado. ¿Cómo se registró el manejador de eventos? ¿Cómo sabe el componente de la GUI que debe llamar a `actionPerformed`, en vez de llamar a otro método manejador de eventos? En la siguiente sección responderemos a estas preguntas y haremos un diagrama de la interacción.

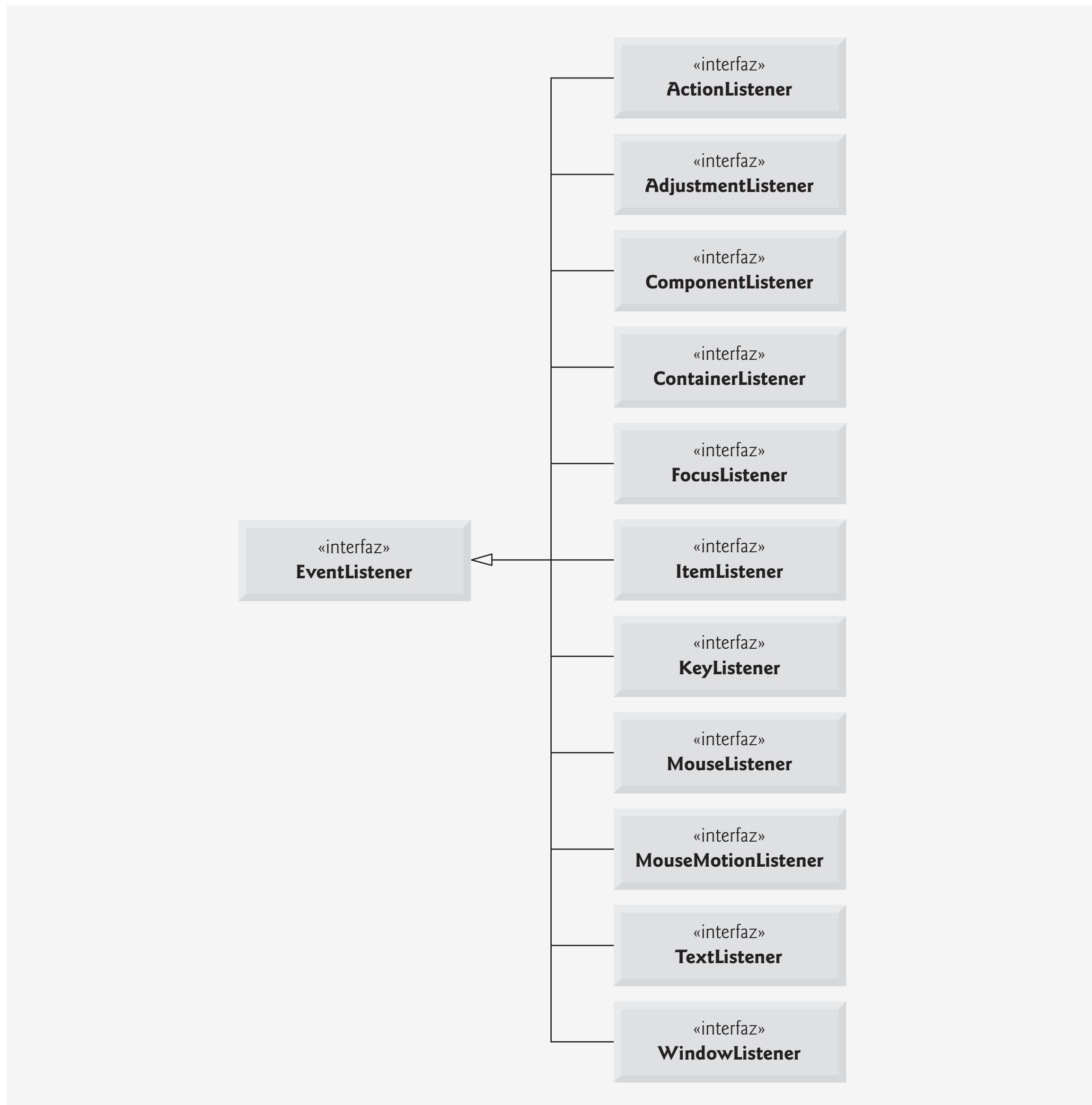


Figura 11.12 | Algunas interfaces comunes de componentes de escucha de eventos del paquete `java.awt.event`.

11.7 Cómo funciona el manejo de eventos

Mostraremos cómo funciona el mecanismo de manejo de eventos, utilizando a `campoTexto1` del ejemplo de la figura 11.9. Tenemos dos preguntas sin contestar de la sección 11.5:

1. ¿Cómo se registró el manejador de eventos?
2. ¿Cómo sabe el componente de la GUI que debe llamar a `actionPerformed` en vez de llamar a algún otro método manejador de eventos?

La primera pregunta se responde mediante el registro de eventos que se lleva a cabo en las líneas 43 a 46 de la aplicación. En la figura 11.3 se muestra un diagrama de la variable `JTextField` llamada `campoTexto1`, la variable `ManejadorCampoTexto` llamada `manejador` y los objetos a los que hacen referencia.

Registro de eventos

Todo `JComponent` tiene una variable de instancia llamada **`listenerList`**, que hace referencia a un objeto de la clase **`EventListenerList`** (paquete `javax.swing.event`). Cada objeto de una subclase de `JComponent` mantiene referencias a todos sus componentes de escucha registrados en `listenerList`. Por cuestión de simpleza, hemos colocado a `listenerList` en el diagrama como un arreglo, abajo del objeto `JTextField` en la figura 11.13.

Cuando se ejecuta la línea 43 de la figura 11.9

```
campoTexto1.addActionListener( manejador );
```

se coloca en el objeto `listenerList` de `campoTexto1` una nueva entrada que contiene una referencia al objeto `ManejadorCampoTexto`. Aunque no se muestra en el diagrama, esta nueva entrada también incluye el tipo del componente de escucha (en este caso, `ActionListener`). Mediante el uso de este mecanismo, cada componente ligero de GUI de Swing mantiene su propia lista de componentes de escucha que se registraron para manejar los eventos del componente.

Invocación del manejador de eventos

El tipo de componente de escucha de eventos es importante para responder a la segunda pregunta: ¿Cómo sabe el componente de la GUI que debe llamar a `actionPerformed` en vez de llamar a otro? Todo componente de la GUI soporta varios tipos de eventos, incluyendo **eventos de ratón**, **eventos de tecla** y otros más. Cuando ocurre un evento, éste se **despacha** solamente a los componentes de escucha de eventos del tipo apropiado. El despachamiento (dispatching) es simplemente el proceso por el cual el componente de la GUI llama a un método manejador de eventos en cada uno de sus componentes de escucha registrados para el tipo de evento que ocurrió.

Cada tipo de evento tiene uno o más interfaces de escucha de eventos correspondientes. Por ejemplo, los eventos tipo `ActionEvent` son manejados por objetos `ActionListener`, los eventos tipo **`MouseEvent`** son manejados por objetos **`MouseListener`** y **`MouseMotionListener`**, y los eventos tipo **`KeyEvent`** son manejados por objetos **`KeyListener`**. Cuando ocurre un evento, el componente de la GUI recibe (de la JVM) un **ID de evento** único, el cual especifica el tipo de evento. El componente de la GUI utiliza el ID de evento para decidir a cuál tipo de componente de escucha debe despacharse el evento, y para decidir cuál método llamar en cada objeto de escucha. Para un `ActionEvent`, el evento se despacha al método `actionPerformed` de todos los objetos `ActionListener` registrados (el único método en la interfaz `ActionListener`). En el caso de un `MouseEvent`, el evento se despacha a todos los objetos `MouseListener` o `MouseMotionListener` registrados, dependiendo del evento de ratón que ocurra. El ID de evento del objeto `MouseListener` determina cuáles de los varios métodos manejadores de eventos de ratón son llamados. Todas estas decisiones las administran los componentes de la GUI

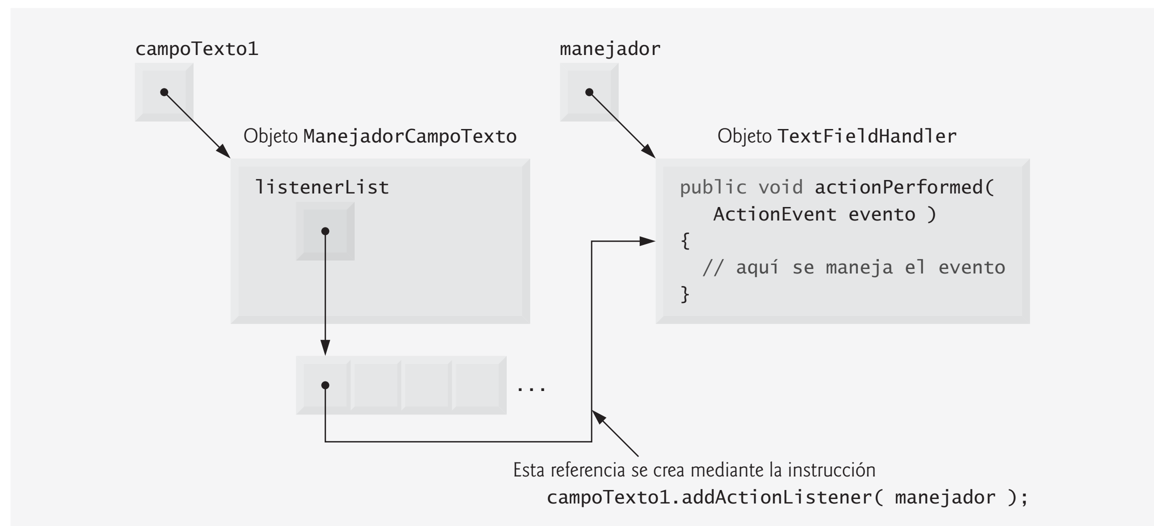


Figura 11.13 | Registro de eventos para el objeto `JTextField` `campoTexto1`.

por usted. Todo lo que usted necesita hacer es registrar un manejador de eventos para el tipo de evento específico que requiere su aplicación, y el componente de GUI asegurará que se llame al método apropiado del manejador de eventos, cuando ocurra el evento. [*Nota:* hablaremos sobre otros tipos de eventos e interfaces de escucha de eventos a medida que se vayan necesitando con cada nuevo componente que vayamos viendo].

11.8 JButton

Un **botón** es un componente en el que el usuario hace clic para desencadenar cierta acción. Una aplicación de Java puede utilizar varios tipos de botones, incluyendo **botones de comando**, **casillas de verificación**, **botones interruptores** y **botones de opción**. En la figura 11.14 se muestra la jerarquía de herencia de los botones de Swing que veremos en este capítulo. Como puede ver en el diagrama, todos los tipos de botones son subclases de **`AbstractButton`** (paquete `javax.swing`), la cual declara las características comunes para los botones de Swing. En esta sección nos concentraremos en los botones que se utilizan comúnmente para iniciar un comando.



Observación de apariencia visual 11.7

Por lo general, los botones utilizan la capitalización estilo título de libro.

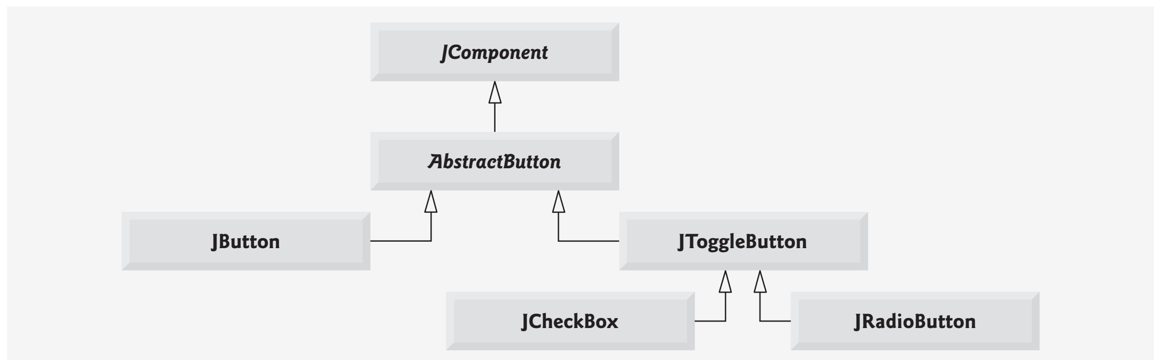


Figura 11.14 | Jerarquía de botones de Swing.

Un botón de comando (vea la salida de la figura 11.15) genera un evento `ActionEvent` cuando el usuario hace clic en él. Los botones de comando se crean con la clase **`JButton`**. El texto en la cara de un objeto `JButton` se llama **etiqueta del botón**. Una GUI puede tener muchos objetos `JButton`, pero cada etiqueta de botón debe generalmente ser única en las partes de la GUI en que se muestre.



Observación de apariencia visual 11.8

Tener más de un objeto `JButton` con la misma etiqueta hace que los objetos `JButton` sean ambiguos para el usuario. Debe proporcionar una etiqueta única para cada botón.

```
// Fig. 11.15: MarcoBoton.java
// Creación de objetos JButton.
import java.awt.FlowLayout;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import javax.swing.JFrame;
import javax.swing.JButton;
import javax.swing.Icon;
import javax.swing.ImageIcon;
```

Figura 11.15 | Botones de comando y eventos de acción. (Parte 1 de 2).

```
10 import javax.swing.JOptionPane;
11
12 public class MarcoBoton extends JFrame
13 {
14    private JButton botonJButtonSimple; // botón con texto solamente
15    private JButton botonJButtonElegante; // botón con iconos
16
17    // MarcoBoton agrega objetos JButton a JFrame
18    public MarcoBoton()
19    {
20       super( "Prueba de botones" );
21       setLayout( new FlowLayout() ); // establece el esquema del marco
22
23       botonJButtonSimple = new JButton( "Boton simple" ); // botón con texto
24       add( botonJButtonSimple ); // agrega botonJButtonSimple a JFrame
25
26       Icon insecto1 = new ImageIcon( getClass().getResource( "insecto1.gif" ) );
27       Icon insecto2 = new ImageIcon( getClass().getResource( "insecto2.gif" ) );
28       botonJButtonElegante = new JButton( "Boton elegante", insecto1 ); // establece la
         imagen
29       botonJButtonElegante.setRolloverIcon( insecto2 ); // establece la imagen de
         sustitución
30       add( botonJButtonElegante ); // agrega botonJButtonElegante a JFrame
31
32       // crea nuevo ManejadorBoton para manejar los eventos de botón
33       ManejadorBoton manejador = new ManejadorBoton();
34       botonJButtonElegante.addActionListener( manejador );
35       botonJButtonSimple.addActionListener( manejador );
36    } // fin del constructor de MarcoBoton
37
38    // clase interna para manejar eventos de botón
39    private class ManejadorBoton implements ActionListener
40    {
41       // maneja evento de botón
42       public void actionPerformed( ActionEvent evento )
43       {
44          JOptionPane.showMessageDialog( MarcoBoton.this, String.format(
45             "Usted oprimio: %s", evento.getActionCommand() ) );
46       } // fin del método actionPerformed
47    } // fin de la clase interna privada ManejadorBoton
48 } // fin de la clase MarcoBoton
```

Figura 11.15 | Botones de comando y eventos de acción. (Parte 2 de 2).

La aplicación de las figuras 11.15 y 11.16 crea dos objetos `JButton` y demuestra que estos objetos tienen soporte para mostrar objetos `Icon`. El manejo de eventos para los botones se lleva a cabo mediante una sola instancia de la clase interna `ManejadorBoton` (líneas 39 a 47).

```
// Fig. 11.16: PruebaBoton.java
// Prueba de MarcoBoton.
import javax.swing.JFrame;

public class PruebaBoton
{
   public static void main( String args[] )
   {
```

Figura 11.16 | Clase de prueba de `MarcoBoton`. (Parte 1 de 2).

```
        MarcoBoton marcoBoton = new MarcoBoton(); // crea MarcoBoton
        marcoBoton.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
        marcoBoton.setSize( 300, 110 ); // establece el tamaño del marco
        marcoBoton.setVisible( true ); // muestra el marco
    } // fin de main
} // fin de la clase PruebaBoton
```

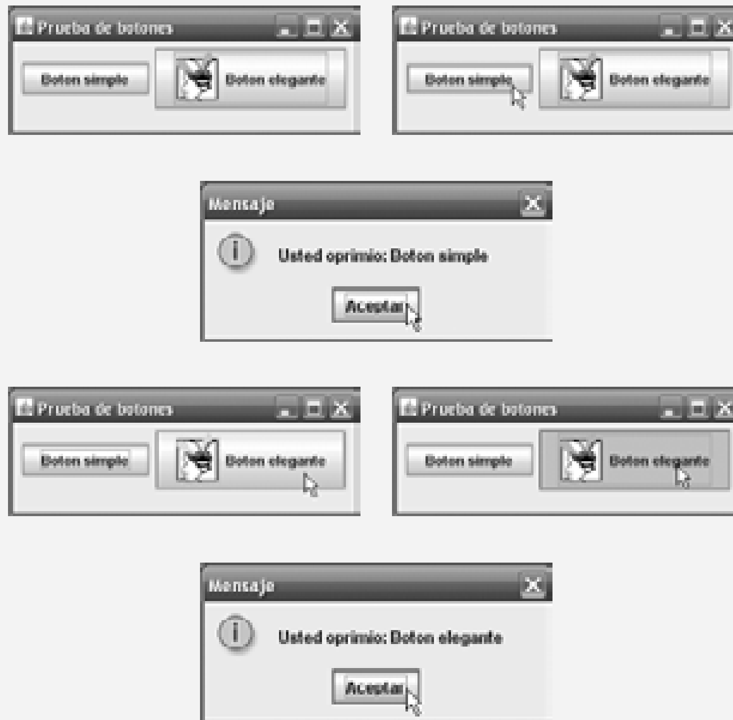


Figura 11.16 | Clase de prueba de `MarcoBoton`. (Parte 2 de 2).

En las líneas 14 y 15 se declaran las variables `botonJButtonSimple` y `botonJButtonElegante` de la clase `JButton`. Los correspondientes objetos se instancian en el constructor. En la línea 23 se crea `botonJButtonSimple` con la etiqueta `"Boton simple"`. En la línea 24 se agrega el botón al objeto `JFrame`.

Un objeto `JButton` puede mostrar un objeto `Icon`. Para proveer al usuario un nivel adicional de interacción visual con la GUI, un objeto `JButton` puede tener también un objeto `Icon` de `sustitución`: un `Icon` que se muestre cuando el usuario coloque el ratón encima del botón. El icono en el botón cambia a medida que el ratón se mueve hacia dentro y fuera del área del botón en la pantalla. En las líneas 26 y 27 se crean dos objetos `ImageIcon` que representan al objeto `Icon` predeterminado y el objeto `Icon` de sustitución para el objeto `JButton` creado en la línea 28. Ambas instrucciones suponen que los archivos de imagen están guardados en el mismo directorio que la aplicación (que es comúnmente el caso para las aplicaciones que utilizan imágenes).

En la línea 28 se crea `botonJButtonElegante` con el texto `"Boton elegante"` y el icono `insecto1`. De manera predeterminada, el texto se muestra a la derecha del icono. En la línea 29 se utiliza el método **`setRolloverIcon`** (heredado de la clase `AbstractButton`) para especificar la imagen a mostrar en el botón cuando el usuario coloque el ratón sobre el botón. En la línea 30 se agrega el botón al objeto `JFrame`.



Observación de apariencia visual 11.9

Como la clase `AbstractButton` soporta el mostrar texto e imágenes en un botón, todas las subclases de `AbstractButton` soportan también el mostrar texto e imágenes.



Observación de apariencia visual 11.10

Al usar iconos de sustitución para los objetos `JButton`*, los usuarios reciben una retroalimentación visual que les indica que, al hacer clic en el ratón mientras el cursor está colocado encima del botón, ocurrirá una acción.*

Los objetos `JButton`, al igual que los objetos `JTextField`, generan eventos `ActionEvent` que pueden ser procesados por cualquier objeto `ActionListener`. En las líneas 33 a 35 se crea un objeto de la clase interna `private ManejadorBoton` y se registra como el manejador de eventos para cada objeto `JButton`. La clase `ManejadorBoton` (líneas 39 a 47) declara a `actionPerformed` para mostrar un cuadro de diálogo de mensaje que contiene la etiqueta del botón que el usuario oprimió. Para un evento de `JButton`, el método `getActionCommand` de `ActionEvent` devuelve la etiqueta del botón.

Cómo acceder a la referencia* `this` *en un objeto de una clase de nivel superior desde una clase interna

Cuando ejecute esta aplicación y haga clic en uno de sus botones, observe que el diálogo de mensaje que aparece está centrado sobre la ventana de la aplicación. Esto ocurre debido a que la llamada al método `showMessageDialog` de `JOptionPane` (líneas 44 y 45 de la figura 11.15) utiliza a `MarcoBoton.this`, en vez de `null` como el primer argumento. Cuando este argumento no es `null`, representa lo que se denomina el componente de GUI padre del diálogo de mensaje (en este caso, la ventana de aplicación es el componente padre) y permite centrar el diálogo sobre ese componente, cuando se muestra el diálogo. `MarcoBoton.this` representa a la referencia `this` del objeto de la clase `MarcoBoton` de nivel superior.



Observación de ingeniería de software 11.4

Cuando se utiliza en una clase interna, la palabra clave `this` *se refiere al objeto actual de la clase interna que se está manipulando. Un método de la clase interna puede utilizar la referencia* `this` *del objeto de la clase externa, si antepone a* `this` *el nombre de la clase externa y un punto, como en* `MarcoBoton.this`.

11.9 Botones que mantienen el estado

Los componentes de la GUI de Swing contienen tres tipos de **botones de estado**: **`JToggleButton`**, **`JCheckBox`** y **`JRadioButton`**, los cuales tienen valores encendido/apagado o verdadero/falso. Las clases `JCheckBox` y `JRadioButton` son subclases de `JToggleButton` (figura 11.14). Un objeto `JRadioButton` es distinto de un objeto `JCheckBox` en cuanto a que generalmente hay varios objetos `JRadioButton` que se agrupan, y son mutuamente exclusivos; sólo uno de los objetos `JRadioButton` en el grupo puede estar seleccionado en un momento dado, de igual forma que los botones en la radio de un automóvil. Primero veremos la clase `JCheckBox`. Las siguientes dos subsecciones también demuestran que una clase interna puede acceder a los miembros de su clase de nivel superior.

11.9.1 JCheckBox

La aplicación de las figuras 11.17 y 11.18 utilizan dos objetos `JCheckBox` para seleccionar el estilo deseado de tipo de letra para el texto a mostrar en un objeto `JTextField`. Un objeto `JCheckBox` aplica un estilo en negritas cuando se selecciona, y el otro aplica un estilo en cursivas. Si ambos se seleccionan, el estilo del tipo de letra es negrita y cursiva. Cuando la aplicación se ejecuta por primera vez, ninguno de los objetos `JCheckBox` está activado (es decir, ambos son `false`), por lo que el tipo de letra es simple. La clase `PruebaCheckBox` (figura 11.18) contiene el método `main` que ejecuta esta aplicación.

```
// Fig. 11.17: MarcoCasillaVerificacion.java
// Creación de botones JCheckBox.
import java.awt.FlowLayout;
import java.awt.Font;
import java.awt.event.ItemListener;
import java.awt.event.ItemEvent;
import javax.swing.JFrame;
import javax.swing.JTextField;
```

Figura 11.17 | Botones `JCheckBox` y eventos de los elementos. (Parte 1 de 2).

```
import javax.swing.JCheckBox;

public class MarcoCasillaVerificacion extends JFrame
{
   private JTextField campoTexto; // muestra el texto en tipos de letra cambiantes
   private JCheckBox negritaJCheckBox; // para seleccionar/deseleccionar negrita
   private JCheckBox cursivaJCheckBox; // para seleccionar/deseleccionar cursiva

   // El constructor de MarcoCasillaVerificacion agrega objetos JCheckBox a JFrame
   public MarcoCasillaVerificacion()
   {
      super( "Prueba de JCheckBox" );
      setLayout( new FlowLayout() ); // establece el esquema del marco

      // establece JTextField y su tipo de letra
      campoTexto = new JTextField( "Observe como cambia el estilo de tipo de letra", 20 );
      campoTexto.setFont( new Font( "Serif", Font.PLAIN, 14 ) );
      add( campoTexto ); // agrega campoTexto a JFrame

      negritaJCheckBox = new JCheckBox( "Negrita" ); // crea casilla de verificación "negrita"
      cursivaJCheckBox = new JCheckBox( "Cursiva" ); // crea casilla de verificación "cursiva"
      add( negritaJCheckBox ); // agrega casilla de verificación "negrita" a JFrame
      add( cursivaJCheckBox ); // agrega casilla de verificación "cursiva" a JFrame

      // registra componentes de escucha para objetos JCheckBox
      ManejadorCheckBox manejador = new ManejadorCheckBox();
      negritaJCheckBox.addItemListener( manejador );
      cursivaJCheckBox.addItemListener( manejador );
   } // fin del constructor de MarcoCasillaVerificacion

   // clase interna privada para el manejo de eventos ItemListener
   private class ManejadorCheckBox implements ItemListener
   {
      private int valNegrita = Font.PLAIN; // controla el estilo de tipo de letra negrita
      private int valCursiva = Font.PLAIN; // controla el estilo de tipo de letra cursiva

      // responde a los eventos de casilla de verificación
      public void itemStateChanged( ItemEvent evento )
      {
         // procesa los eventos de la casilla de verificación "negrita"
         if ( evento.getSource() == negritaJCheckBox )
            valNegrita =
               negritaJCheckBox.isSelected() ? Font.BOLD : Font.PLAIN;

         // procesa los eventos de la casilla de verificación "cursiva"
         if ( evento.getSource() == cursivaJCheckBox )
            valCursiva =
               cursivaJCheckBox.isSelected() ? Font.ITALIC : Font.PLAIN;

         // establece el tipo de letra del campo de texto
         campoTexto.setFont(
            new Font( "Serif", valNegrita + valCursiva, 14 ) );
      } // fin del método itemStateChanged
   } // fin de la clase interna privada ManejadorCheckBox
} // fin de la clase MarcoCasillaVerificacion
```

Figura 11.17 | Botones `JCheckBox` y eventos de los elementos. (Parte 2 de 2).

```
// Fig. 11.18: PruebaCasillaVerificacion.java
// Prueba de MarcoCasillaVerificacion.
import javax.swing.JFrame;

public class PruebaCasillaVerificacion
{
   public static void main( String args[] )
   {
      MarcoCasillaVerificacion marcoCasillaVerificacion = new MarcoCasillaVerificacion();
      marcoCasillaVerificacion.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
      marcoCasillaVerificacion.setSize( 350, 100 ); // establece el tamaño del marco
      marcoCasillaVerificacion.setVisible( true ); // muestra el marco
   } // fin de main
} // fin de la clase PruebaCasillaVerificacion
```



Figura 11.18 | Clase de prueba de `MarcoCasillaVerificacion`.

Una vez creado e inicializado el objeto `JTextField` (figura 11.17, línea 24), en la línea 25 se utiliza el método **`setFont`** (heredado por `JTextField` indirectamente de la clase `Component`) para establecer el tipo de letra del objeto `JTextField` con un nuevo objeto de la clase **`Font`** (paquete `java.awt`). El nuevo objeto `Font` se inicializa con `"Serif"` (un nombre de tipo de letra genérico que representa un tipo de letra como `Times`, y se soporta en todas las plataformas de Java), estilo `Font.PLAIN` y tamaño de 14 puntos. A continuación, en las líneas 28 y 29 se crean dos objetos `JCheckBox`. La cadena que se pasa al constructor de `JCheckBox` es la **etiqueta de la casilla de verificación** que aparece a la derecha del objeto `JCheckBox` de manera predeterminada.

Cuando el usuario hace clic en un objeto `JCheckBox`, ocurre un evento **`ItemEvent`**. Este evento puede manejarse mediante un objeto **`ItemListener`**, que debe implementar al método **`itemStateChanged`**. En este ejemplo, el manejo de eventos se lleva a cabo mediante una instancia de la clase interna `private ManejadorCasillaVerificacion` (líneas 40 a 62). En las líneas 34 a 36 se crea una instancia de la clase `ManejadorCasillaVerificacion` y se registra con el método **`addItemListener`** como componente de escucha para ambos objetos `JCheckBox`.

En las líneas 42 y 43 se declaran variables de instancia para la clase interna `ManejadorCasillaVerificacion`. En conjunto, estas variables representan el estilo de tipo de letra para el texto que se muestra en el objeto `JTextField`. Al principio ambas son `Font.PLAIN` para indicar que el tipo de letra no es negrita y no es cursiva. El método `itemStateChanged` (líneas 46 a 61) es llamado cuando el usuario hace clic en el objeto `JCheckBox negrita` o `cursiva`. Este método utiliza `evento.getSource()` para determinar en cuál de los objetos `JCheckBox` se hizo clic. Si fue en la casilla `negritaJCheckBox`, en la línea 51 se utiliza el método **`isSelected`** de `JCheckBox` para determinar si el botón está seleccionado (es decir, marcado). Si es así, a la variable local `valNegrita` se le asigna `Font.BOLD`; en caso contrario, se le asigna `Font.PLAIN`. Una instrucción similar se ejecuta si el usuario hace clic en `cursivaJCheckBox`. Si esta casilla de verificación está seleccionada, a la variable local `valCursiva` se le asigna `Font.ITALIC`; en caso contrario, se le asigna `Font.PLAIN`. En las líneas 59 y 60 se cambia el tipo de letra del objeto `JTextField`, usando el mismo nombre de tipo de letra y tamaño de punto. La suma de `valNegrita` y `valCursiva` representa el nuevo estilo de tipo de letra del objeto `JTextField`. Cada una de las constantes de `Font` representa un valor único. `Font.PLAIN` tiene el valor `0`, por lo que si tanto `valNegrita` como `valCursiva` se establecen en `Font.PLAIN`, el tipo de letra tendrá el estilo simple. Si uno de los valores es `Font.BOLD` o `Font.ITALIC`, el tipo de letra estará en negrita o en cursiva, respectivamente. Si uno es `BOLD` y el otro es `ITALIC`, el tipo de letra estará en negrita y en cursiva.

Relación entre una clase interna y su clase de nivel superior

Tal vez haya observado que la clase `ManejadorCasillaVerificacion` utilizó las variables `negritaJCheckBox` (figura 11.17, líneas 49 y 51), `cursivaJCheckBox` (líneas 54 y 56) y `campoTexto` (línea 59), aun cuando estas variables no se declaran en la clase interna. Una clase interna tiene una relación especial con su clase de nivel superior; a la clase interna se le permite acceder directamente a todas las variables de instancia y métodos de la clase de nivel superior. El método `itemStateChanged` (líneas 46 a 61) de la clase `ManejadorCasillaVerificacion` utiliza esta relación para determinar cuál objeto `JCheckBox` es el origen del evento, para determinar el estado de un objeto `JCheckBox` y para establecer el tipo de letra en el objeto `JTextField`. Observe que ninguna parte del código en la clase interna `ManejadorCasillaVerificacion` requiere una referencia al objeto de la clase de nivel superior.

11.9.2 JRadioButton

Los **botones de opción** (que se declaran con la clase **`JRadioButton`**) son similares a las casillas de verificación, en cuanto a que tienen dos estados: seleccionado y no seleccionado (al que también se le conoce como deseleccionado). Sin embargo, los botones de opción generalmente aparecen como un **grupo**, en el cual sólo un botón de opción puede estar seleccionado en un momento dado (vea la salida de la figura 11.20). Al seleccionar un botón de opción distinto en el grupo se obliga a que todos los demás botones de opción del grupo se deseleccionen. Los botones de opción se utilizan para representar un conjunto de opciones **mutuamente exclusivas** (es decir, no pueden seleccionarse varias opciones en el grupo al mismo tiempo). La relación lógica entre los botones de opción se mantiene mediante un objeto **`ButtonGroup`** (paquete `javax.swing`), el cual en sí no es un componente de la GUI. Un objeto `ButtonGroup` organiza un grupo de botones y no se muestra a sí mismo en una interfaz de usuario. En vez de ello, se muestra en la GUI cada uno de los objetos `JRadioButton` del grupo.



Error común de programación 11.3

Si se agrega un objeto `ButtonGroup` (o un objeto de cualquier otra clase que no se derive de `Component`) a un contenedor, se produce un error de compilación.

La aplicación de las figuras 11.19 y 11.20 es similar a la de las figuras 1.17 y 11.18. El usuario puede alterar el estilo del tipo de letra del texto de un objeto `JTextField`. La aplicación utiliza botones de opción que permiten que se seleccione solamente un estilo de tipo de letra en el grupo a la vez. La clase `PruebaBotonOpcion` (figura 11.20) contiene el método `main` que ejecuta esta aplicación.

```
// Fig. 11.19: MarcoBotonOpcion.java
// Creación de botones de opción, usando ButtonGroup y JRadioButton.
import java.awt.FlowLayout;
import java.awt.Font;
import java.awt.event.ItemListener;
import java.awt.event.ItemEvent;
import javax.swing.JFrame;
import javax.swing.JTextField;
import javax.swing.JRadioButton;
import javax.swing.ButtonGroup;

public class MarcoBotonOpcion extends JFrame
{
   private JTextField campoTexto; // se utiliza para mostrar los cambios en el tipo de
   letra
   private Font tipoLetraSimple; // tipo de letra para texto simple
   private Font tipoLetraNegrita; // tipo de letra para texto en negrita
   private Font tipoLetraCursiva; // tipo de letra para texto en cursiva
   private Font tipoLetraNegritaCursiva; // tipo de letra para texto en negrita y
   cursiva
```

Figura 11.19 | Objetos `JRadioButton` y `ButtonGroup`. (Parte 1 de 3).

```
   private JRadioButton simpleJRadioButton; // selecciona texto simple
   private JRadioButton negritaJRadioButton; // selecciona texto en negrita
   private JRadioButton cursivaJRadioButton; // selecciona texto en cursiva
   private JRadioButton negritaCursivaJRadioButton; // negrita y cursiva
   private ButtonGroup grupoOpciones; // grupo de botones que contiene los botones de
   opción

   // El constructor de MarcoBotonOpcion agrega los objetos JRadioButton a JFrame
   public MarcoBotonOpcion()
   {
      super( "Prueba de RadioButton" );
      setLayout( new FlowLayout() ); // establece el esquema del marco

      campoTexto = new JTextField( "Observe el cambio en el estilo del tipo de letra",
      28 );
      add( campoTexto ); // agrega campoTexto a JFrame

      // crea los botones de opción
      simpleJRadioButton = new JRadioButton( "Simple", true );
      negritaJRadioButton = new JRadioButton( "Negrita", false );
      cursivaJRadioButton = new JRadioButton( "Cursiva", false );
      negritaCursivaJRadioButton = new JRadioButton( "Negrita/Cursiva", false );
      add( simpleJRadioButton ); // agrega botón simple a JFrame
      add( negritaJRadioButton ); // agrega botón negrita a JFrame
      add( cursivaJRadioButton ); // agrega botón cursiva a JFrame
      add( negritaCursivaJRadioButton ); // agrega botón negrita y cursiva

      // crea una relación lógica entre los objetos JRadioButton
      grupoOpciones = new ButtonGroup(); // crea ButtonGroup
      grupoOpciones.add( simpleJRadioButton ); // agrega simple al grupo
      grupoOpciones.add( negritaJRadioButton ); // agrega negrita al grupo
      grupoOpciones.add( cursivaJRadioButton ); // agrega cursiva al grupo
      grupoOpciones.add( negritaCursivaJRadioButton ); // agrega negrita y cursiva

      // crea objetos tipo de letra
      tipoLetraSimple = new Font( "Serif", Font.PLAIN, 14 );
      tipoLetraNegrita = new Font( "Serif", Font.BOLD, 14 );
      tipoLetraCursiva = new Font( "Serif", Font.ITALIC, 14 );
      tipoLetraNegritaCursiva = new Font( "Serif", Font.BOLD + Font.ITALIC, 14 );
      campoTexto.setFont( tipoLetraSimple ); // establece tipo letra inicial a simple

      // registra eventos para los objetos JRadioButton
      simpleJRadioButton.addItemListener(
         new ManejadorBotonOpcion( tipoLetraSimple ) );
      negritaJRadioButton.addItemListener(
         new ManejadorBotonOpcion( tipoLetraNegrita ) );
      cursivaJRadioButton.addItemListener(
         new ManejadorBotonOpcion( tipoLetraCursiva ) );
      negritaCursivaJRadioButton.addItemListener(
         new ManejadorBotonOpcion( tipoLetraNegritaCursiva ) );
   } // fin del constructor de MarcoBotonOpcion

   // clase interna privada para manejar eventos de botones de opción
   private class ManejadorBotonOpcion implements ItemListener
   {
      private Font tipoLetra; // tipo de letra asociado con este componente de escucha

      public ManejadorBotonOpcion( Font f )
      {
```

Figura 11.19 | Objetos `JRadioButton` y `ButtonGroup`. (Parte 2 de 3).

```
76            tipoLetra = f; // establece el tipo de letra de este componente de escucha
77         } // fin del constructor ManejadorBotonOpcion
78
79         // maneja los eventos de botones de opción
80         public void itemStateChanged( ItemEvent evento )
81         {
82            campoTexto.setFont( tipoLetra ); // establece el tipo de letra de campoTexto
83         } // fin del método itemStateChanged
84      } // fin de la clase interna privada ManejadorBotonOpcion
85   } // fin de la clase MarcoBotonOpcion
```

Figura 11.19 | Objetos `JRadioButton` y `ButtonGroup`. (Parte 3 de 3).

```
// Fig. 11.20: PruebaBotonOpcion.java
// Prueba de MarcoBotonOpcion.
import javax.swing.JFrame;

public class PruebaBotonOpcion
{
   public static void main( String args[] )
   {
      MarcoBotonOpcion marcoBotonOpcion = new MarcoBotonOpcion();
      marcoBotonOpcion.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
      marcoBotonOpcion.setSize( 350, 100 ); // establece el tamaño del marco
      marcoBotonOpcion.setVisible( true ); // muestra el marco
   } // fin de main
} // fin de la clase PruebaBotonOpcion
```



Figura 11.20 | Clase de prueba de `MarcoBotonOpcion`.

En las líneas 35 a 42 del constructor (figura 11.19) se crean cuatro objetos `JRadioButton` y se agregan al objeto `JFrame`. Cada objeto `JRadioButton` se crea con una llamada al constructor como la de la línea 35. Este constructor especifica la etiqueta que aparece a la derecha del objeto `JRadioButton` de manera predeterminada, junto con su estado inicial. Un segundo argumento `true` indica que el objeto `JRadioButton` debe aparecer seleccionado al mostrarlo en pantalla.

En la línea 45 se instancia un objeto `ButtonGroup` llamado `grupoOpciones`. Este objeto es el "pegamento" que forma la relación lógica entre los cuatro objetos `JRadioButton` y permite que se seleccione solamente uno de los cuatro en un momento dado. Es posible que no se seleccione ningún `JRadioButton` en un `ButtonGroup`, pero esto sólo puede ocurrir si no se agregan objetos `JRadioButton` preseleccionados al objeto `ButtonGroup`, y si el usuario no ha seleccionado todavía un objeto `JRadioButton`. En las líneas 46 a 49 se utiliza el método **`add`** de `ButtonGroup` para asociar cada uno de los objetos `JRadioButton` con `grupoOpciones`. Si se agrega al grupo más de un objeto `JRadioButton` seleccionado, el primer objeto `JRadioButton` seleccionado que se agregue será el que quede seleccionado cuando se muestre la GUI en pantalla.

Los objetos `JRadioButton`, al igual que los objetos `JCheckbox`, generan eventos tipo `ItemEvent` cuando se hace clic sobre ellos. En las líneas 59 a 66 se crean cuatro instancias de la clase interna `ManejadorBotonOpcion` (declarada en las líneas 70 a 84). En este ejemplo, cada objeto componente de escucha de eventos se registra para manejar el evento `ItemEvent` que se genera cuando el usuario hace clic en cualquiera de los objetos `JRadioButton`. Observe que cada objeto `ManejadorBotonOpcion` se inicializa con un objeto `Font` específico (creado en las líneas 52 a 55).

La clase `ManejadorBotonOpcion` (línea 70 a 84) implementa la interfaz `ItemListener` para poder manejar los eventos `ItemEvent` generados por los objetos `JRadioButton`. El constructor almacena el objeto `Font` que recibe como un argumento en la variable de instancia `tipoLetra` (declarada en la línea 72) del objeto componente de escucha de eventos. Cuando el usuario hace clic en un objeto `JRadioButton`, `grupoOpciones` desactiva el objeto `JRadioButton` previamente seleccionado y el método `itemStateChanged` (líneas 80 a 83) establece el tipo de letra en el objeto `JTextField` al tipo de letra almacenado en el objeto componente de escucha de eventos correspondiente al objeto `JRadioButton`. Observe que la línea 82 de la clase interna `ManejadorBotonesOpcion` utiliza la variable de instancia `campoTexto` de la clase de nivel superior para establecer el tipo de letra.

11.10 JComboBox y el uso de una clase interna anónima para el manejo de eventos

Un cuadro combinado (algunas veces conocido como **lista desplegable**) proporciona una lista de elementos (figura 11.22), de la cual el usuario puede seleccionar solamente uno. Los cuadros combinados se implementan con la clase **`JComboBox`**, la cual extiende a la clase `JComponent`. Los objetos `JComboBox` generan eventos `ItemEvent`, al igual que los objetos `JCheckBox` y `JRadioButton`. Este ejemplo también demuestra una forma especial de clase interna, que se utiliza con frecuencia en el manejo de eventos.

La aplicación de las figuras 11.21 y 11.22 utiliza un objeto `JComboBox` para proporcionar una lista de cuatro nombres de archivos de imágenes, de los cuales el usuario puede seleccionar una imagen para mostrarla en pantalla. Cuando el usuario selecciona un nombre, la aplicación muestra la imagen correspondiente como un objeto `Icon` en un objeto `JLabel`. La clase `PruebaCuadroCombinado` (figura 11.22) contiene el método `main` que ejecuta esta aplicación. Las capturas de pantalla para esta aplicación muestran la lista `JComboBox` después de hacer una selección, para ilustrar cuál nombre de archivo de imagen fue seleccionado.

```java
// Fig. 11.21: MarcoCuadroCombinado.java
// Uso de un objeto JComboBox para seleccionar una imagen a mostrar.
import java.awt.FlowLayout;
import java.awt.event.ItemListener;
import java.awt.event.ItemEvent;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JComboBox;
import javax.swing.Icon;
import javax.swing.ImageIcon;

public class MarcoCuadroCombinado extends JFrame
{
   private JComboBox imagenesJComboBox; // cuadro combinado con los nombres de los
   iconos
   private JLabel etiqueta; // etiqueta para mostrar el icono seleccionado

   private String nombres[] =
      { "insecto1.gif", "insecto2.gif", "insectviaje.gif", "insectanim.gif" };
   private Icon iconos[] = {
      new ImageIcon( getClass().getResource( nombres[ 0 ] ) ),
      new ImageIcon( getClass().getResource( nombres[ 1 ] ) ),
      new ImageIcon( getClass().getResource( nombres[ 2 ] ) ),
      new ImageIcon( getClass().getResource( nombres[ 3 ] ) ) };
```

Figura 11.21 | Objeto `JComboBox` que muestra una lista de nombres de imágenes. (Parte 1 de 2).

```

   // El constructor de MarcoCuadroCombinado agrega un objeto JComboBox a JFrame
   public MarcoCuadroCombinado()
   {
      super( "Prueba de JComboBox" );
      setLayout( new FlowLayout() ); // establece el esquema del marco

      imagenesJComboBox = new JComboBox( nombres ); // establece JComboBox
      imagenesJComboBox.setMaximumRowCount( 3 ); // muestra tres filas

      imagenesJComboBox.addItemListener(
         new ItemListener() // clase interna anónima
         {
            // maneja evento de JComboBox
            public void itemStateChanged( ItemEvent evento )
            {
               // determina si está seleccionada la casilla de verificación
               if ( evento.getStateChange() == ItemEvent.SELECTED )
                  etiqueta.setIcon( iconos[
                     imagenesJComboBox.getSelectedIndex() ] );
            } // fin del método itemStateChanged
         } // fin de la clase interna anónima
      ); // fin de la llamada a addItemListener

      add( imagenesJComboBox ); // agrega cuadro combinado a JFrame
      etiqueta = new JLabel( iconos[ 0 ] ); // muestra el primer icono
      add( etiqueta ); // agrega etiqueta a JFrame
   } // fin del constructor de MarcoCuadroCombinado
} // fin de la clase MarcoCuadroCombinado
```

Figura 11.21 | Objeto `JComboBox` que muestra una lista de nombres de imágenes. (Parte 2 de 2).

```
// Fig. 11.22: PruebaCuadroCombinado.java
// Prueba de MarcoCuadroCombinado.
import javax.swing.JFrame;

public class PruebaCuadroCombinado
{
   public static void main( String args[] )
   {
      MarcoCuadroCombinado marcoCuadroCombinado = new MarcoCuadroCombinado();
      marcoCuadroCombinado.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
      marcoCuadroCombinado.setSize( 350, 150 ); // establece el tamaño del marco
      marcoCuadroCombinado.setVisible( true ); // muestra el marco
   } // fin de main
} // fin de la clase PruebaCuadroCombinado
```

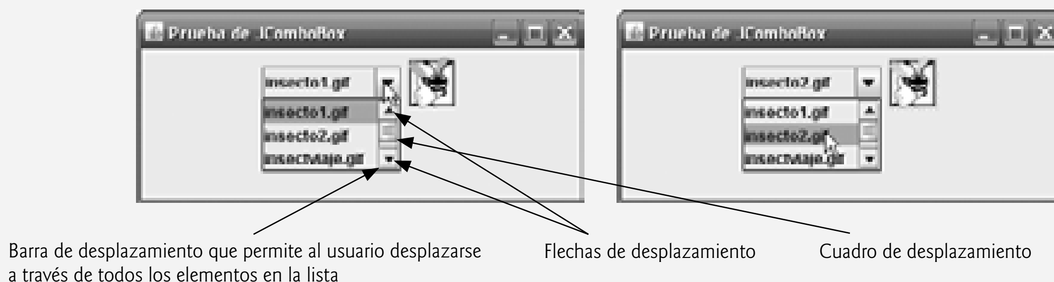


Figura 11.22 | Clase de prueba de `MarcoCuadroCombinado`. (Parte 1 de 2).



Figura 11.22 | Clase de prueba de `MarcoCuadroCombinado`. (Parte 2 de 2).

En las líneas 19 a 23 (figura 11.21) se declara e inicializa el arreglo `iconos` con cuatro nuevos objetos `ImageIcon`. El arreglo `String` llamado `nombres` (líneas 17 y 18) contiene los nombres de los cuatro archivos de imagen que se guardan en el mismo directorio que la aplicación.

En la línea 31 se crea un objeto `JComboBox`, utilizando los objetos `String` en el arreglo `nombres` como los elementos en la lista. Cada elemento de la lista tiene un **índice**. El primer elemento se agrega en el índice 0; el siguiente elemento se agrega en el índice 1, y así sucesivamente. El primer elemento que se agrega a un objeto `JComboBox` aparece como el elemento actualmente seleccionado al mostrar el objeto `JComboBox`. Los otros elementos se seleccionan haciendo clic en el objeto `JComboBox`, el cual se expande en una lista de la cual el usuario puede hacer una selección.

En la línea 32 se utiliza el método **`setMaximumRowCount`** de `JComboBox` para establecer el máximo número de elementos a mostrar cuando el usuario haga clic en el objeto `JComboBox`. Si hay elementos adicionales, el objeto `JComboBox` proporciona una **barra de desplazamiento** (vea la primera captura de pantalla) que permite al usuario desplazarse por todos los elementos en la lista. El usuario puede hacer clic en las **flechas de desplazamiento** que están en las partes superior e inferior de la barra de desplazamiento para avanzar hacia arriba y hacia debajo de la lista, un elemento a la vez, o puede arrastrar hacia arriba y hacia abajo el **cuadro de desplazamiento** que está en medio de la barra de desplazamiento para desplazarse por la lista. Para arrastrar el cuadro de desplazamiento, mantenga presionado el botón izquierdo del ratón mientras éste se encuentra sobre el cuadro de desplazamiento, y mueva el ratón.



Observación de apariencia visual 11.11

Establezca el número máximo de filas en un objeto JComboBox a un valor que evite que la lista se expanda fuera de los límites de la ventana o subprograma en la que se utilice. Esta configuración asegurará que la lista se muestre correctamente cuando el usuario la expanda.

La línea 48 adjunta el objeto `JComboBox` al esquema `FlowLayout` de `MarcoCuadroCombinado` (que se establece en la línea 29). La línea 49 crea el objeto `JLabel` que muestra objetos `ImageIcon` y lo inicializa con el primer objeto `ImageIcon` en el arreglo `iconos`. La línea 50 adjunta el objeto `JLabel` al esquema `FlowLayout` de `MarcoCuadroCombinado`.

Uso de una clase interna anónima para el manejo de eventos

Las líneas 34 a 46 son una instrucción que declara la clase del componente de escucha de eventos, crea un objeto de esa clase y registra el objeto como el componente de escucha para los eventos `ItemEvent` de `imagenesJComboBox`. En este ejemplo, el objeto componente de escucha de eventos es una instancia de una **clase interna anónima**; una forma especial de clase interna que se declara sin un nombre y, por lo general, aparece dentro de la declaración de un método. Al igual que las demás clases internas, una clase interna anónima puede acceder a los miembros de su clase de nivel superior. Sin embargo, una clase interna anónima tiene acceso limitado a las variables locales del método en el que está declarada. Como una clase interna anónima no tiene nombre, un objeto de la clase interna anónima debe crearse en el punto en el que se declara la clase (empezando en la línea 35).



Observación de ingeniería de software 11.5

Una clase interna anónima declarada en un método puede acceder a las variables de instancia y los métodos del objeto de la clase de nivel superior que la declaró, así como a las variables locales `final` *del método, pero no puede acceder a las variables locales no* `final` *del método.*

Las líneas 34 a 46 son una llamada al método `addItemListener` de `imagenesJComboBox`. El argumento para este método debe ser un objeto que *sea un* `ItemListener` (es decir, cualquier objeto de una clase que implemente a `ItemListener`). Las líneas 35 a 45 son una expresión de creación de instancias de clase que declara una clase interna anónima y crea un objeto de esa clase. Después se pasa una referencia a ese objeto como argumento para `addItemListener`. La sintaxis `ItemListener()` después de `new` empieza la declaración de una clase interna anónima que implementa a la interfaz `ItemListener`. Esto es similar a empezar una declaración con

```
public class MiManejador implements ItemListener
```

Los paréntesis después de `ItemListener` indican una llamada al constructor predeterminado de la clase interna anónima.

La llave izquierda de apertura (`{`) en la línea 36 y la llave derecha de cierre (`}`) en la línea 45 delimitan el cuerpo de la clase interna anónima. Las líneas 38 a 44 declaran el método `itemStateChanged` de `ItemListener`. Cuando el usuario hace una selección de `imagenesJComboBox`, este método establece el objeto `Icon` de `etiqueta`. El objeto `Icon` se selecciona del arreglo `iconos`, determinando el índice del elemento seleccionado en el objeto `JComboBox` con el método **`getSelectedIndex`** en la línea 43. Observe que para cada elemento seleccionado de un `JComboBox`, primero se deselecciona otro elemento; por lo tanto, ocurren dos eventos tipo `ItemEvent` cuando se selecciona un elemento. Deseamos mostrar sólo el icono para el elemento que el usuario acaba de seleccionar. Por esta razón, la línea 41 determina si el método **`getStateChange`** de `ItemEvent` devuelve `ItemEvent.SELECTED`. De ser así, las líneas 42 y 43 establecen el icono de `etiqueta`.



Observación de ingeniería de software 11.6

Al igual que cualquier otra clase, cuando una clase interna anónima implementa a una interfaz, la clase debe implementar todos los métodos en la interfaz.

La sintaxis que se muestra en las líneas 35 a 45 para crear un manejador de eventos con una clase interna anónima es similar al código que genera un entorno de desarrollo integrado (IDE) de Java. Por lo general, un IDE permite al programador diseñar una GUI en forma visual, y después el IDE genera código que implementa a la GUI. El programador sólo inserta instrucciones en los métodos manejadores de eventos que declaran cómo manejar cada evento.

11.11 JList

Una *lista* muestra una serie de elementos, de la cual el usuario puede seleccionar uno o más (vea la salida de la figura 11.23). Las listas se crean con la clase `JList`, que extiende directamente a la clase `JComponent`. La clase `JList` soporta **listas de selección simple** (listas que permiten seleccionar solamente un elemento a la vez) y **listas de selección múltiple** (listas que permiten seleccionar cualquier número de elementos a la vez). En esta sección hablaremos sobre las listas de selección simple.

La aplicación de las figuras 11.23 y 11.24 crea un objeto `JList` que contiene los nombres de 13 colores. Al hacer clic en el nombre de un color en el objeto `JList`, ocurre un evento **`ListSelectionEvent`** y la aplicación cambia el color de fondo de la ventana de aplicación al color seleccionado. La clase `PruebaLista` (figura 11.24) contiene el método `main` que ejecuta esta aplicación.

```
// Fig. 11.23: MarcoLista.java
// Selección de colores de un objeto JList.
import java.awt.FlowLayout;
import java.awt.Color;
import javax.swing.JFrame;
import javax.swing.JList;
import javax.swing.JScrollPane;
import javax.swing.event.ListSelectionListener;
import javax.swing.event.ListSelectionEvent;
import javax.swing.ListSelectionModel;
```

Figura 11.23 | Objeto `JList` que muestra una lista de colores. (Parte 1 de 2).

```
11
12 public class MarcoLista extends JFrame
13 {
14    private JList listaJListColores; // lista para mostrar colores
15    private final String nombresColores[] = { "Negro", "Azul", "Cyan",
16       "Gris oscuro", "Gris", "Verde", "Gris claro", "Magenta",
17       "Naranja", "Rosa", "Rojo", "Blanco", "Amarillo" };
18    private final Color colores[] = { Color.BLACK, Color.BLUE, Color.CYAN,
19       Color.DARK_GRAY, Color.GRAY, Color.GREEN, Color.LIGHT_GRAY,
20       Color.MAGENTA, Color.ORANGE, Color.PINK, Color.RED, Color.WHITE,
21       Color.YELLOW };
22
23    // El constructor de MarcoLista agrega a JFrame el JScrollPane que contiene a JList
24    public MarcoLista()
25    {
26       super( "Prueba de JList" );
27       setLayout( new FlowLayout() ); // establece el esquema del marco
28
29       listaJListColores = new JList( nombresColores ); // crea con nombresColores
30       listaJListColores.setVisibleRowCount( 5 ); // muestra cinco filas a la vez
31
32       // no permite selecciones múltiples
33       listaJListColores.setSelectionMode( ListSelectionModel.SINGLE_SELECTION );
34
35       // agrega al marco un objeto JScrollPane que contiene a JList
36       add( new JScrollPane( listaJListColores ) );
37
38       listaJListColores.addListSelectionListener(
39          new ListSelectionListener() // clase interna anónima
40          {
41             // maneja los eventos de selección de la lista
42             public void valueChanged( ListSelectionEvent evento )
43             {
44                getContentPane().setBackground(
45                   colores[ listaJListColores.getSelectedIndex() ] );
46             } // fin del método valueChanged
47          } // fin de la clase interna anónima
48       ); // fin de la llamada a addListSelectionListener
49    } // fin del constructor de MarcoLista
50 } // fin de la clase MarcoLista
```

Figura 11.23 | Objeto `JList` que muestra una lista de colores. (Parte 2 de 2).

La línea 29 (figura 11.23) crea el objeto `listaJListColores` llamado `JList`. El argumento para el constructor de `JList` es el arreglo de objetos `Object` (en este caso, objetos `String`) a mostrar en la lista. La línea 30 utiliza el método **`setVisibleRowCount`** de `JList` para determinar el número de elementos visibles en la lista.

La línea 33 utiliza el método **`setSelectionMode`** de `JList` para especificar el **modo de selección** de la lista. La clase **`ListSelectionModel`** (del paquete `javax.swing`) declara tres constantes que especifican el modo de selección de un objeto `JList`: **`SINGLE_SELECTION`** (que sólo permite seleccionar un elemento a la vez), **`SINGLE_INTERVAL_SELECTION`** (para una lista de selección múltiple que permite seleccionar varios elementos contiguos) y **`MULTIPLE_INTERVAL_SELECTION`** (para una lista de selección múltiple que no restringe los elementos que se pueden seleccionar).

A diferencia de un objeto `JComboBox`, un objeto `JList` *no* proporciona una barra de desplazamiento si hay más elementos en la lista que el número de filas visibles. En este caso se utiliza un objeto **`JScrollPane`** para proporcionar la capacidad de desplazamiento. En la línea 36 se agrega una nueva instancia de la clase `JScrollPane` al objeto `JFrame`. El constructor de `JScrollPane` recibe como argumento el objeto `JComponent` que necesita funcionalidad de desplazamiento (en este caso, `listaJListColores`). Observe en las capturas de pantalla que aparece una barra de desplazamiento creada por el objeto `JScrollPane` en el lado derecho del objeto `JList`. De

```java
// Fig. 11.24: PruebaLista.java
// Selección de colores de un objeto JList.
import javax.swing.JFrame;

public class PruebaLista
{
   public static void main( String args[] )
   {
      MarcoLista marcoLista = new MarcoLista(); // crea objeto MarcoLista
      marcoLista.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
      marcoLista.setSize( 350, 150 ); // establece el tamaño del marco
      marcoLista.setVisible( true ); // muestra el marco
   } // fin de main
} // fin de la clase PruebaLista
```



Figura 11.24 | Clase de prueba de `MarcoLista`.

manera predeterminada, la barra de desplazamiento sólo aparece cuando el número de elementos en el objeto `JList` excede al número de elementos visibles.

Las líneas 38 a 48 usan el método **`addListSelectionListener`** de `JList` para registrar un objeto que implementa a **`ListSelectionListener`** (paquete `javax.swing.event`) como el componente de escucha para los eventos de selección de `JList`. Una vez más, utilizamos una instancia de una clase interna anónima (líneas 39 a 47) como el componente de escucha. En este ejemplo, cuando el usuario realiza una selección de `listaJListColores`, el método **`valueChanged`** (línea 42 a 46) debería cambiar el color de fondo del objeto `MarcoLista` al color seleccionado. Esto se logra en las líneas 44 y 45. Observe el uso del método **`getContentPane`** de `JFrame` en la línea 44. Cada objeto `JFrame` en realidad consiste de tres niveles: el fondo, el panel de contenido y el panel de vidrio. El panel de contenido aparece en frente del fondo, y es en donde se muestran los componentes de la GUI en el objeto `JFrame`. El panel de vidrio se utiliza para mostrar cuadros de información sobre herramientas y otros elementos que deben aparecer enfrente de los componentes de la GUI en la pantalla. El panel de contenido oculta por completo el fondo del objeto `JFrame`; por ende, para cambiar el color de fondo detrás de los componentes de la GUI, debe cambiar el color de fondo del panel de contenido. El método `getContentPane` devuelve una referencia al panel de contenido del objeto `JFrame` (un objeto de la clase `Container`). En la línea 44, después usamos esa referencia para llamar al método **`setBackground`**, el cual establece el color de fondo del panel de contenido a un elemento en el arreglo `colores`. El color se selecciona del arreglo mediante el uso del índice del elemento seleccionado. El método **`getSelectedItem`** de `JList` devuelve el índice del elemento seleccionado. Al igual que con los arreglos y los objetos `JComboBox`, los índices en los objetos `JList` están basados en cero.

11.12 Listas de selección múltiple

Una **lista de selección múltiple** permite al usuario seleccionar varios elementos de un objeto `JList` (vea la salida de la figura 11.26). Una lista `SINGLE_INTERVAL_SELECTION` permite la selección de un rango contiguo de elementos. Para ello, haga clic en el primer elemento y después oprima (y mantenga oprimida) la tecla *Mayús* mientras hace clic en el último elemento a seleccionar en el rango. Una lista `MULTIPLE_INTERVAL_SELECTION` permite una selección de rango continuo, como se describe para una lista `SINGLE_INTERVAL_SELECTION`. Dicha lista permite que se seleccionen diversos elementos, oprimiendo y manteniendo oprimida la tecla *Ctrl* (a la que algunas veces se le conoce como la tecla de *Control*) mientras hace clic en cada elemento a seleccionar. Para deseleccionar un elemento, oprima y mantenga oprimida la tecla *Ctrl* mientras hace clic en el elemento por segunda vez.

La aplicación de las figuras 11.25 y 11.26 utiliza listas de selección múltiple para copiar elementos de un objeto `JList` a otro. Una lista es de tipo `MULTIPLE_INTERVAL_SELECTION` y la otra es de tipo `SINGLE_INTERVAL_SELECTION`. Cuando ejecute la aplicación, trate de usar las técnicas de selección descritas anteriormente para seleccionar elementos en ambas listas.

```
// Fig. 11.25: MarcoSeleccionMultiple.java
// Copiar elementos de un objeto List a otro.
import java.awt.FlowLayout;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import javax.swing.JFrame;
import javax.swing.JList;
import javax.swing.JButton;
import javax.swing.JScrollPane;
import javax.swing.ListSelectionModel;

public class MarcoSeleccionMultiple extends JFrame
{
   private JList listaJListColores; // lista para guardar los nombres de los colores
   private JList listaJListCopia; // lista en la que se van a copiar los nombres de los colores
   private JButton botonJButtonCopiar; // botón para copiar los nombres seleccionados
   private final String nombresColores[] = { "Negro", "Azul", "Cyan",
      "Gris oscuro", "Gris", "Verde", "Gris claro", "Magenta", "Naranja",
      "Rosa", "Rojo", "Blanco", "Amarillo"};

   // Constructor de MarcoSeleccionMultiple
   public MarcoSeleccionMultiple()
   {
      super( "Listas de seleccion multiple" );
      setLayout( new FlowLayout() ); // establece el esquema del marco

      listaJListColores = new JList( nombresColores ); // contiene nombres de todos los colores
      listaJListColores.setVisibleRowCount( 5 ); // muestra cinco filas
      listaJListColores.setSelectionMode(
         ListSelectionModel.MULTIPLE_INTERVAL_SELECTION );
      add( new JScrollPane( listaJListColores ) ); // agrega lista con panel de desplazamiento

      botonJButtonCopiar = new JButton( "Copiar >>>" ); // crea botón para copiar
      botonJButtonCopiar.addActionListener(

         new ActionListener() // clase interna anónima
         {
            // maneja evento de botón
            public void actionPerformed( ActionEvent evento )
            {
               // coloca los valores seleccionados en listaJListCopia
               listaJListCopia.setListData( listaJListColores.getSelectedValues() );
            } // fin del método actionPerformed
         } // fin de la clase interna anónima
      ); // fin de la llamada a addActionListener

      add( botonJButtonCopiar ); // agrega el botón copiar a JFrame

```

Figura 11.25 | Objeto `JList` que permite selecciones múltiples. (Parte 1 de 2).

```
49         listaJListCopia = new JList(); // crea lista para guardar nombres de colores
           copiados
50         listaJListCopia.setVisibleRowCount( 5 ); // muestra 5 filas
51         listaJListCopia.setFixedCellWidth( 100 ); // establece la anchura
52         listaJListCopia.setFixedCellHeight( 15 ); // establece la altura
53         listaJListCopia.setSelectionMode(
54            ListSelectionModel.SINGLE_INTERVAL_SELECTION );
55         add( new JScrollPane( listaJListCopia ) ); // agrega lista con panel de
           desplazamiento
56      } // fin del constructor de MarcoSeleccionMultiple
57   } // fin de la clase MarcoSeleccionMultiple
```

Figura 11.25 | Objeto `JList` que permite selecciones múltiples. (Parte 2 de 2).

```
// Fig. 11.26: PruebaSeleccionMultiple.java
// Prueba de MarcoSeleccionMultiple.
import javax.swing.JFrame;

public class PruebaSeleccionMultiple
{
   public static void main( String args[] )
   {
      MarcoSeleccionMultiple marcoSeleccionMultiple =
         new MarcoSeleccionMultiple();
      marcoSeleccionMultiple.setDefaultCloseOperation(
         JFrame.EXIT_ON_CLOSE );
      marcoSeleccionMultiple.setSize( 350, 140 ); // establece el tamaño del marco
      marcoSeleccionMultiple.setVisible( true ); // muestra el marco
   } // fin de main
} // fin de la clase PruebaSeleccionMultiple
```



Figura 11.26 | Clase de prueba de `MarcoSeleccionMultiple`.

En la línea 27 de la figura 11.25 se crea el objeto `JList` llamado `listaJListColores` y se inicializa con las cadenas en el arreglo `nombresColores`. En la línea 28 se establece el número de filas visibles en `listaJListColores` a 5. En las líneas 29 y 30 se especifica que `listaJListColores` es una lista de tipo `MULTIPLE_INTERVAL_SELECTION`. En la línea 31 se agrega un nuevo objeto `JScrollPane`, que contiene `listaJListColores`, al panel `JFrame`. En las líneas 49 a 55 se realizan tareas similares para `listaJListCopia`, la cual se declara como una lista tipo `SINGLE_INTERVAL_SELECTION`. En la línea 51 se utiliza el método **`setFixedCellWidth`** de `JList` para establecer la anchura de `listaJListCopia` en 100 píxeles. En la línea 52 se utiliza el método **`setFixedCellHeight`** de `JList` para establecer la altura de cada elemento en el objeto `JList` a 15 píxeles.

Una lista de selección múltiple no tiene eventos para indicar que un usuario ha realizado varias selecciones. Normalmente, un evento generado por otro componente de la GUI (lo que se conoce como un **evento externo**) especifica cuándo deben procesarse las selecciones múltiples en un objeto `JList`. En este ejemplo, el usuario hace clic en el objeto `JButton` llamado `botonJButtonCopiar` para desencadenar el evento que copia los elementos seleccionados en `listaJListColores` a `listaJListCopia`.

Las líneas 39 a 45 declaran, crean y registran un objeto `ActionListener` para el objeto `botonJButtonCopiar`. Cuando el usuario hace clic en `botonJButtonCopiar`, el método `actionPerformed` (líneas 39 a 43) utiliza el método **`setListData`** de `JList` para establecer los elementos mostrados en `listaJListCopia`. En la línea 42 se hace una llamada al método **`getSelectedValues`** de `listaJListColores`, el cual devuelve un arreglo de objetos `Object` que representan los elementos seleccionados en `listaJListColores`. En este ejemplo, el arreglo devuelto se pasa como argumento al método `setListData` de `listaJListCopia`.

Tal vez se pregunte por qué puede usarse `listaJListCopia` en la línea 42, aun cuando la aplicación no crea el objeto al cual hace referencia sino hasta la línea 49. Recuerde que el método `actionPerformed` (líneas 39 a 43) no se ejecuta sino hasta que el usuario oprime el botón `botonJButtonCopiar`, lo cual no puede ocurrir sino hasta que el constructor termine su ejecución y la aplicación muestre la GUI. En ese punto en la ejecución de la aplicación, `listaJListCopia` ya se ha inicializado con un nuevo objeto `JList`.

11.13 Manejo de eventos de ratón

En esta sección presentaremos las interfaces de escucha de eventos **`MouseListener`** y **`MouseMotionListener`** para manejar **eventos de ratón**. Estos eventos pueden atraparse para cualquier componente de la GUI que se derive de `java.awt.Component`. Los métodos de las interfaces `MouseListener` y `MouseMotionListener` se sintetizan en la figura 11.27. El paquete `javax.swing.event` contiene la interfaz **`MouseInputListener`**, la cual extiende a las interfaces `MouseListener` y `MouseMotionListener` para crear una sola interfaz que contiene todos los métodos de `MouseListener` y `MouseMotionListener`. Estos métodos se llaman cuando el ratón interactúa con un objeto `Component`, si se registran objetos componentes de escucha de eventos para ese objeto `Component`.

Métodos de las interfaces `MouseListener` y `MouseMotionListener`

Métodos de la interfaz `MouseListener`

`public void mousePressed( MouseEvent evento )`

Se llama cuando se oprime un botón del ratón, mientras el cursor del ratón está sobre un componente.

`public void mouseClicked( MouseEvent evento )`

Se llama cuando se oprime y suelta un botón del ratón, mientras el cursor del ratón permanece estacionario sobre un componente. Este evento siempre va precedido por una llamada a `mousePressed`.

`public void mouseReleased( MouseEvent evento )`

Se llama cuando se suelta un botón de ratón después de ser oprimido. Este evento siempre va precedido por una llamada a `mousePressedy` por una o más llamadas a `mouseDragged`.

`public void mouseEntered( MouseEvent evento )`

Se llama cuando el cursor del ratón entra a los límites de un componente.

`public void mouseExited( MouseEvent evento )`

Se llama cuando el cursor del ratón sale de los límites de un componente.

Métodos de la interfaz `MouseMotionListener`

`public void mouseDragged( MouseEvent evento )`

Se llama cuando el botón del ratón se oprime mientras el cursor del ratón se encuentra sobre un componente y se mueve mientras el botón sigue oprimido. Este evento siempre va precedido por una llamada a `mousePressed`. Todos los eventos de arrastre del ratón se envían al componente en el cual empezó la acción de arrastre.

`public void mouseMoved( MouseEvent evento )`

Se llama al moverse el ratón cuando su cursor se encuentra sobre un componente. Todos los eventos de movimiento se envían al componente sobre el cual se encuentra el ratón posicionado en ese momento.

Figura 11.27 | Métodos de las interfaces `MouseListener` y `MouseMotionListener`.

Cada uno de los métodos manejadores de eventos de ratón toma un objeto **`MouseEvent`** como su argumento. Un objeto `MouseEvent` contiene información acerca del evento de ratón que ocurrió, incluyendo las coordenadas *x* y *y* de la ubicación en donde ocurrió el evento. Estas coordenadas se miden desde la esquina superior izquierda del componente de la GUI en el que ocurrió el evento. Las coordenadas *x* empiezan en 0 y se incrementan de izquierda a derecha. Las coordenadas *y* empiezan en 0 y se incrementan de arriba hacia abajo. Además, los métodos y constantes de la clase **`InputEvent`** (superclase de `MouseEvent`) permiten a una aplicación determinar cuál fue el botón del ratón que oprimió el usuario.



Observación de apariencia visual 11.12

Las llamadas a los métodos `mouseDragged` y `mouseReleased` se envían al objeto `MouseMotionListener` para el objeto `Component` en el que empezó la operación de arrastre. De manera similar, la llamada al método `mouseReleased` al final de una operación de arrastre se envía al objeto `MouseListener` para el objeto `Component` en el que empezó la operación de arrastre.

Java también cuenta con la interfaz **`MouseWheelListener`** para permitir a las aplicaciones responder a la rotación del disco en un ratón que tenga uno. Esta interfaz declara el método **`mouseWheelMoved`**, el cual recibe un evento **`MouseWheelEvent`** como argumento. La clase `MouseWheelEvent` (una subclase de `MouseEvent`) contiene métodos que permiten al manejador de eventos obtener información acerca de la cantidad de rotación del disco.

Cómo rastrear eventos de ratón en un objeto `JPanel`

La aplicación `RastreadorRaton` (figuras 11.28 y 11.29) demuestra el uso de los métodos de las interfaces `MouseListener` y `MouseMotionListener`. La clase de aplicación implementa ambas interfaces, para poder escuchar sus propios eventos de ratón. Observe que los siete métodos de estas dos interfaces deben ser declarados por el programador cuando una clase implementa ambas interfaces. Cada evento de ratón en este ejemplo muestra una cadena en el objeto `JLabel` llamado `barraEstado`, en la parte inferior de la ventana.

```java
// Fig. 11.28: MarcoRastreadorRaton.java
// Demostración de los eventos de ratón.
import java.awt.Color;
import java.awt.BorderLayout;
import java.awt.event.MouseListener;
import java.awt.event.MouseMotionListener;
import java.awt.event.MouseEvent;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;

public class MarcoRastreadorRaton extends JFrame
{
   private JPanel panelRaton; // panel en el que ocurrirán los eventos de ratón
   private JLabel barraEstado; // etiqueta que muestra información de los eventos

   // El constructor de MarcoRastreadorRaton establece la GUI y
   // registra los manejadores de eventos de ratón
   public MarcoRastreadorRaton()
   {
      super( "Demostracion de los eventos de raton" );

      panelRaton = new JPanel(); // crea el panel
      panelRaton.setBackground( Color.WHITE ); // establece el color de fondo
      add( panelRaton, BorderLayout.CENTER ); // agrega el panel a JFrame

      barraEstado = new JLabel( "Raton fuera de JPanel" );
      add( barraEstado, BorderLayout.SOUTH ); // agrega etiqueta a JFrame
```

Figura 11.28 | Manejo de eventos de ratón. (Parte 1 de 3).

```

      // crea y registra un componente de escucha para los eventos de ratón y de su movimiento
      ManejadorRaton manejador = new ManejadorRaton();
      panelRaton.addMouseListener( manejador );
      panelRaton.addMouseMotionListener( manejador );
   } // fin del constructor de MarcoRastreadorRaton

   private class ManejadorRaton implements MouseListener,
      MouseMotionListener
   {
      // Los manejadores de eventos de MouseListener
      // manejan el evento cuando se suelta el ratón justo después de oprimir el botón
      public void mouseClicked( MouseEvent evento )
      {
         barraEstado.setText( String.format( "Se hizo clic en [%d, %d]",
            evento.getX(), evento.getY() ) );
      } // fin del método mouseClicked

      // maneja evento cuando se oprime el ratón
      public void mousePressed( MouseEvent evento )
      {
         barraEstado.setText( String.format( "Se oprimio en [%d, %d]",
            evento.getX(), evento.getY() ) );
      } // fin del método mousePressed

      // maneja evento cuando se suelta el botón del ratón después de arrastrarlo
      public void mouseReleased( MouseEvent evento )
      {
         barraEstado.setText( String.format( "Se solto en [%d, %d]",
            evento.getX(), evento.getY() ) );
      } // fin del método mouseReleased

      // maneja evento cuando el ratón entra al área
      public void mouseEntered( MouseEvent evento )
      {
         barraEstado.setText( String.format( "Raton entro en [%d, %d]",
            evento.getX(), evento.getY() ) );
         panelRaton.setBackground( Color.GREEN );
      } // fin del método mouseEntered

      // maneja evento cuando el ratón sale del área
      public void mouseExited( MouseEvent evento )
      {
         barraEstado.setText( "Raton fuera de JPanel" );
         panelRaton.setBackground( Color.WHITE );
      } // fin del método mouseExited

      // Los manejadores de eventos de MouseMotionListener manejan
      // el evento cuando el usuario arrastra el ratón con el botón oprimido
      public void mouseDragged( MouseEvent evento )
      {
         barraEstado.setText( String.format( "Se arrastro en [%d, %d]",
            evento.getX(), evento.getY() ) );
      } // fin del método mouseDragged

      // maneja evento cuando el usuario mueve el ratón
      public void mouseMoved( MouseEvent evento )
      {
```

Figura 11.28 | Manejo de eventos de ratón. (Parte 2 de 3).

```
            barraEstado.setText( String.format( "Se movio en [%d, %d]",
               evento.getX(), evento.getY() ) );
         } // fin del método mouseMoved
      } // fin de la clase interna ManejadorRaton
   } // fin de la clase MarcoRastreadorRaton
```

Figura 11.28 | Manejo de eventos de ratón. (Parte 3 de 3).

```
// Fig. 11.29: MarcoRastreadorRaton.java
// Prueba de MarcoRastreadorRaton.
import javax.swing.JFrame;

public class RastreadorRaton
{
   public static void main( String args[] )
   {
      MarcoRastreadorRaton marcoRastreadorRaton = new MarcoRastreadorRaton();
      marcoRastreadorRaton.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
      marcoRastreadorRaton.setSize( 300, 100 ); // establece el tamaño del marco
      marcoRastreadorRaton.setVisible( true ); // muestra el marco
   } // fin de main
} // fin de la clase RastreadorRaton
```



Figura 11.29 | Clase de prueba de `MarcoRastreadorRaton`.

La línea 23 en la figura 11.28 crea el objeto `JPanel` llamado `panelRaton`. Los eventos de ratón de este objeto `JPanel` serán rastreados por la aplicación. En la línea 24 se establece el color de fondo de `panelRaton` a blanco. Cuando el usuario mueva el ratón hacia el `panelRaton`, la aplicación cambiará el color de fondo de `panelRaton` a verde. Cuando el usuario mueva el ratón hacia fuera del `panelRaton`, la aplicación cambiará el color de fondo de vuelta a blanco. En la línea 25 se adjunta el objeto `panelRaton` al objeto `JFrame`. Como vimos en la sección 11.4, por lo general, debemos especificar el esquema de los componentes de GUI en un objeto `JFrame`. En esa sección presentamos el administrador de esquemas `FlowLayout`. Aquí utilizamos el esquema predeterminado del panel de contenido de un objeto `JFrame`: **`BorderLayout`**. Este administrador de esquemas ordena los componentes en cinco regiones: **`NORTH`**, **`SOUTH`**, **`EAST`**, **`WEST`** y **`CENTER`**. `NORTH` corresponde a la parte superior del contenedor. Este ejemplo utiliza las regiones `CENTER` y `SOUTH`. En la línea 25 se utiliza una versión con dos argumentos del método `add` para colocar a `panelRaton` en la región `CENTER`. El esquema `BorderLayout` ajusta automáticamente el tamaño del componente en la región `CENTER` para utilizar todo el espacio en el objeto `JFrame` que no esté ocupado por los componentes de otras regiones. En la sección 11.17.2 hablaremos sobre `BorderLayout` con más detalle.

En las líneas 27 y 28 del constructor se declara el objeto `JLabel` llamado `barraEstado` y se adjunta a la región `SOUTH` del objeto `JFrame`. Este objeto `JLabel` ocupa la anchura del objeto `JFrame`. La altura de la región se determina en base al objeto `JLabel`.

En la línea 31 se crea una instancia de la clase interna `ManejadorRaton` (líneas 36 a 90) llamada `manejador`, la cual responde a los eventos de ratón. En las líneas 32 y 33 se registra `manejador` como el componente de escucha para los eventos de ratón de `panelRaton`. Los métodos **`addMouseListener`** y **`addMouseMotionListener`** se heredan indirectamente de la clase `Component`, y pueden utilizarse para registrar objetos `MouseListener` y `MouseMotionListener`, respectivamente. Un objeto `ManejadorRaton` es tanto un `MouseListener` como un `MotionListener`, ya que la clase implementa ambas interfaces. [*Nota:* en este ejemplo, optamos por implementar ambas interfaces para demostrar una clase que implementa más de una interfaz. Sin embargo, también pudimos haber implementado la interfaz `MouseInputListener` aquí].

Cuando el ratón entra y sale del área de `panelRaton`, se hacen llamadas a los métodos `mouseEntered` (líneas 62 a 67) y `mouseExited` (líneas 70 a 74), respectivamente. El método `mouseEntered` muestra un mensaje en el objeto `barraEstado`, indicando que el ratón entró al objeto `JPanel` y cambia el color de fondo a verde. El método `mouseExited` muestra un mensaje en el objeto `barraEstado`, indicando que el ratón está fuera del objeto `JPanel` (vea la primera ventana de resultados de ejemplo) y cambia el color de fondo a blanco.

Cuando ocurre cualquiera de los otros cinco eventos, se muestra un mensaje en el objeto `barraEstado` que incluye una cadena, la cual contiene el evento y las coordenadas en las que ocurrió. Los métodos **`getX`** y **`getY`** de `MouseEvent` devuelven las coordenadas *x* y *y*, respectivamente, del ratón en el momento en el que ocurrió el evento.

11.14 Clases adaptadoras

Muchas de las interfaces de escucha de eventos, como `MouseListener` y `MouseMotionListener`, contienen varios métodos. No siempre es deseable declarar todos los métodos en una interfaz de escucha de eventos. Por ejemplo, una aplicación podría necesitar solamente el manejador `mouseClicked` de la interfaz `MouseListener`, o el manejador `mouseDragged` de la interfaz `MouseMotionListener`. La interfaz `WindowListener` especifica siete métodos manejadores de eventos. Para muchas de las interfaces de escucha de eventos que contienen varios métodos, el paquete `java.awt.event` y el paquete `javax.swing.event` proporcionan clases adaptadoras de escucha de eventos. Una **clase adaptadora** implementa a una interfaz y proporciona una implementación predeterminada (con un cuerpo vacío para los métodos) de todos los métodos en la interfaz. En la figura 11.30 se muestran varias clases adaptadoras de `java.awt.event`, junto con las interfaces que implementan. Usted puede extender una clase adaptadora para heredar la implementación predeterminada de cada método, y en consecuencia sobrescribir sólo el(los) método(s) que necesite para manejar eventos.



Observación de ingeniería de software 11.7

Cuando una clase implementa a una interfaz, la clase tiene una relación del tipo "es un" *con esa interfaz. Todas las subclases directas e indirectas de esa clase heredan esta interfaz. Por lo tanto, un objeto de una clase que extiende a una clase adaptadora de eventos es un objeto del tipo de escucha de eventos correspondiente (por ejemplo, un objeto de una subclase de* `MouseAdapter` *es un* `MouseListener`*).*

Clase adaptadora de eventos en `java.awt.event`	Implementa a la interfaz
`ComponentAdapter`	`ComponentListener`
`ContainerAdapter`	`ContainerListener`
`FocusAdapter`	`FocusListener`
`KeyAdapter`	`KeyListener`
`MouseAdapter`	`MouseListener`
`MouseMotionAdapter`	`MouseMotionListener`
`WindowAdapter`	`WindowListener`

Figura 11.30 | Las clases adaptadoras de eventos y las interfaces que implementan en el paquete `java.awt.event`.

Extensión de `MouseAdapter`

La aplicación de las figuras 11.31 y 11.32 demuestra cómo determinar el número de clics del ratón (es decir, la cuenta de clics) y cómo diferenciar los distintos botones del ratón. El componente de escucha de eventos en esta aplicación es un objeto de la clase interna `ManejadorClicRaton` (líneas 26 a 46) que extiende a `MouseAdapter`, por lo que podemos declarar sólo el método `mouseClicked` que necesitamos en este ejemplo.



Error común de programación 11.4

Si extiende una clase adaptadora y escribe de manera incorrecta el nombre del método que está sobrescribiendo, su método simplemente se vuelve otro método en la clase. Éste es un error lógico difícil de detectar, ya que el programa llamará a la versión vacía del método heredado de la clase adaptadora.

```
1  // Fig. 11.31: MarcoDetallesRaton.java
2  // Demostración de los clics del ratón y cómo diferenciar los botones del mismo.
3  import java.awt.BorderLayout;
4  import java.awt.Graphics;
5  import java.awt.event.MouseAdapter;
6  import java.awt.event.MouseEvent;
7  import javax.swing.JFrame;
8  import javax.swing.JLabel;
9
10 public class MarcoDetallesRaton extends JFrame
11 {
12    private String detalles; // objeto String que representa al objeto JLabel
13    private JLabel barraEstado; // que aparece en la parte inferior de la ventana
14
15    // constructor establece String de la barra de título y registra componente de
      escucha del ratón
16    public MarcoDetallesRaton()
17    {
18       super( "Clics y botones del raton");
19
20       barraEstado = new JLabel( "Haga clic en el raton" );
21       add( barraEstado, BorderLayout.SOUTH );
22       addMouseListener( new ManejadorClicRaton() ); // agrega el manejador
23    } // fin del constructor de MarcoDetallesRaton
24
25    // clase interna para manejar los eventos del ratón
26    private class ManejadorClicRaton extends MouseAdapter
27    {
28       // maneja evento de clic del ratón y determina cuál botón se oprimió
29       public void mouseClicked( MouseEvent evento )
30       {
31          int xPos = evento.getX(); // obtiene posición x del ratón
32          int yPos = evento.getY(); // obtiene posición y del ratón
33
34          detalles = String.format( "Se hizo clic %d vez(veces)",
35             evento.getClickCount() );
36
37          if ( evento.isMetaDown() ) // botón derecho del ratón
38             detalles += " con el boton derecho del raton";
39          else if ( evento.isAltDown() ) // botón central del ratón
40             detalles += " con el boton central del raton";
41          else // botón izquierdo del ratón
42             detalles += " con el boton izquierdo del raton";
43
```

Figura 11.31 | Clics de los botones izquierdo, central y derecho del ratón. (Parte 1 de 2).

```
            barraEstado.setText( detalles ); // muestra mensaje en barraEstado
         } // fin del método mouseClicked
      } // fin de la clase interna privada ManejadorClicRaton
   } // fin de la clase MarcoDetallesRaton
```

Figura 11.31 | Clics de los botones izquierdo, central y derecho del ratón. (Parte 2 de 2).

```
// Fig. 11.32: DetallesRaton.java
// Prueba de MarcoDetallesRaton.
import javax.swing.JFrame;

public class DetallesRaton
{
   public static void main( String args[] )
   {
      MarcoDetallesRaton marcoDetallesRaton = new MarcoDetallesRaton();
      marcoDetallesRaton.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
      marcoDetallesRaton.setSize( 400, 150 ); // establece el tamaño del marco
      marcoDetallesRaton.setVisible( true ); // muestra el marco
   } // fin de main
} // fin de la clase DetallesRaton
```



Figura 11.32 | Clase de prueba de `MarcoDetallesRaton`.

Un usuario de una aplicación en Java puede estar en un sistema con un ratón de uno, dos o tres botones. Java cuenta con un mecanismo para diferenciar cada uno de los botones del ratón. La clase `MouseEvent` hereda varios métodos de la clase `InputEvent` que pueden diferenciar los botones del ratón en un ratón con varios botones, o pueden imitar un ratón de varios botones con una combinación de teclas y un clic del botón del ratón. La figura 11.33 muestra los métodos de `InputEvent` que se utilizan para diferenciar los clics de los botones del ratón. Java asume que cada ratón contiene un botón izquierdo del ratón. Por ende, es fácil probar un clic del botón izquierdo del ratón. Sin embargo, los usuarios con un ratón de uno o dos botones deben usar una combinación de teclas y clics con el botón del ratón al mismo tiempo, para simular los botones que éste no tenga. En el caso de un ratón con uno o dos botones, una aplicación de Java asume que se hizo clic en el botón central del ratón, si el usuario mantiene oprimida la tecla *Alt* y hace clic en el botón izquierdo en un ratón con dos botones, o el único botón en un ratón con un botón. En el caso de un ratón con un botón, una aplicación de Java asume que se hizo clic en el botón derecho si el usuario mantiene oprimida la tecla *Meta* y hace clic en el botón del ratón.

La línea 22 de la figura 11.31 registra un objeto `MouseListener` para el `MarcoDetallesRaton`. El componente de escucha de eventos es un objeto de la clase `ManejadorClicRaton`, el cual extiende a `MouseAdapter`. Esto nos permite declarar sólo el método `mouseClicked` (líneas 29 a 45). Este método primero captura las coordenadas en donde ocurrió el evento y las almacena en las variables locales `xPos` y `yPos` (líneas 31 y 32). Las líneas 34 y 35 crean un objeto `String` llamado `detalles` que contiene el número de clics del ratón, el cual se devuelve mediante el método **`getClickCount`** de `MouseEvent` en la línea 35. Las líneas 37 a 42 utilizan los métodos **`isMetaDown`** e **`isAltDown`** para determinar cuál botón del ratón oprimió el usuario, y adjuntan un objeto `String` apropiado a `detalles` en cada caso. El objeto `String` resultante se muestra en la `barraEstado`. La clase `DetallesRaton` (figura 11.32) contiene el método `main` que ejecuta la aplicación. Pruebe haciendo clic con cada uno de los botones de su ratón repetidas veces, para ver el incremento en la cuenta de clics.

Método InputEvent	Descripción
`isMetaDown()`	Devuelve `true` cuando el usuario hace clic en el botón derecho del ratón, en un ratón con dos o tres botones. Para simular un clic con el botón derecho del ratón en un ratón con un botón, el usuario puede mantener oprimida la tecla *Meta* en el teclado y hacer clic con el botón del ratón.
`isAltDown()`	Devuelve `true` cuando el usuario hace clic con el botón central del ratón, en un ratón con tres botones. Para simular un clic con el botón central del ratón en un ratón con uno o dos botones, el usuario puede oprimir la tecla *Alt* en el teclado y hacer clic en el único botón o en el botón izquierdo del ratón, respectivamente.

Figura 11.33 | Métodos de `InputEvent` que ayudan a diferenciar los clics de los botones izquierdo, central y derecho del ratón.

11.15 Subclase de JPanel para dibujar con el ratón

La sección 11.13 mostró cómo rastrear los eventos del ratón en un objeto `JPanel`. En esta sección usaremos un objeto `JPanel` como un **área dedicada de dibujo**, en la cual el usuario puede dibujar arrastrando el ratón. Además, esta sección demuestra un componente de escucha de eventos que extiende a una clase adaptadora.

Método `paintComponent`

Los componentes ligeros de Swing que extienden a la clase `JComponent` (como `JPanel`) contienen el método **`paintComponent`**, el cual se llama cuando se muestra un componente ligero de Swing. Al sobrescribir este método, puede especificar cómo dibujar figuras usando las herramientas de gráficos de Java. Al personalizar un objeto `JPanel` para usarlo como un área dedicada de dibujo, la subclase debe sobrescribir el método `paintComponent` y llamar a la versión de `paintComponent` de la superclase como la primera instrucción en el cuerpo del método sobrescrito, para asegurar que el componente se muestre en forma correcta. La razón de ello es que las subclases de `JComponent` soportan la **transparencia**. Para mostrar un componente en forma correcta, el programa debe determinar si el componente es transparente. El código que determina esto en la implementación del método `paintComponent` de la superclase `JComponent`. Cuando un componente es transparente, `paintComponent` no borra su fondo cuando el programa muestra el componente. Cuando un componente es **opaco**, `paintComponent` borra el fondo del componente antes de mostrarlo. Si no se hace una llamada a la versión de `paintComponent` de la superclase, por lo general, un componente de GUI opaco no se mostrará correctamente en la interfaz de usuario. Además, si se hace una llamada a la versión de la superclase después de realizar las instrucciones de dibujo personalizadas, por lo general, se borran los resultados. La transparencia de un componente ligero de Swing puede establecerse con el método **`setOpaque`** (un argumento `false` indica que el componente es transparente).



Observación de apariencia visual 11.13

La mayoría de los componentes de GUI pueden ser transparentes u opacos. Si un componente de GUI de Swing es opaco, su fondo se borrará cuando se haga una llamada a su método `paintComponent`. Sólo los componentes opacos pueden mostrar un color de fondo personalizado. Los objetos `JPanel` son opacos de manera predeterminada.



Tip para prevenir errores 11.1

En el método `paintComponent` *de una subclase de* `JComponent`*, la primera instrucción siempre debe ser una llamada al método* `paintComponent` *de la superclase, para asegurar que un objeto de la subclase se muestre en forma correcta.*



Error común de programación 11.5

Si un método `paintComponent` *sobrescrito no llama a la versión de la superclase, el componente de la subclase tal vez no se muestre en forma apropiada. Si un método* `paintComponent` *sobrescrito llama a la versión de la superclase después de realizar otro dibujo, éste se borra.*

Definición del área de dibujo personalizada

La aplicación `Pintor` de las figuras 11.34 y 11.35 demuestra una **subclase personalizada de `JPanel`** que se utiliza para crear un área dedicada de dibujo. La aplicación utiliza el manejador de eventos `mouseDragged` para crear una aplicación simple de dibujo. El usuario puede dibujar imágenes arrastrando el ratón en el objeto `JPanel`. Este ejemplo no utiliza el método `mouseMoved`, por lo que nuestra clase de escucha de eventos (la clase interna anónima en las líneas 22 a 34) extiende a `MouseMotionAdapter`. Como esta clase ya declara tanto a `mouseMoved` como `mouseDragged`, simplemente podemos sobrescribir a `mouseDragged` para proporcionar el manejo de eventos que requiere esta aplicación.

La clase `PanelDibujo` (figura 11.34) extiende a `JPanel` para crear el área dedicada de dibujo. Las líneas 3 a 7 importan las clases que se utilizan en la clase `PanelDibujo`. La clase **`Point`** (paquete `java.awt`) representa una coordenada *x-y*. Utilizamos objetos de esta clase para almacenar las coordenadas de cada evento de arrastre del ratón. La clase **`Graphics`** se utiliza para dibujar.

```
1  // Fig. 11.34: PanelDibujo.java
2  // Uso de la clase MouseMotionAdapter.
3  import java.awt.Point;
4  import java.awt.Graphics;
5  import java.awt.event.MouseEvent;
6  import java.awt.event.MouseMotionAdapter;
7  import javax.swing.JPanel;
8  
9  public class PanelDibujo extends JPanel
10 {
11    private int cuentaPuntos = 0; // cuenta el número de puntos
12 
13    // arreglo de 10000 referencias a java.awt.Point
14    private Point puntos[] = new Point[ 10000 ];
15 
16    // establece la GUI y registra el manejador de eventos del ratón
17    public PanelDibujo()
18    {
19       // maneja evento de movimiento del ratón en el marco
20       addMouseMotionListener(
21 
22          new MouseMotionAdapter() // clase interna anónima
23          {
24             // almacena las coordenadas de arrastre y vuelve a dibujar
25             public void mouseDragged( MouseEvent evento )
26             {
27                if ( cuentaPuntos < puntos.length )
28                {
29                   puntos[ cuentaPuntos ] = evento.getPoint(); // busca el punto
30                   cuentaPuntos++; // incrementa el número de puntos en el arreglo
31                   repaint(); // vuelve a dibujar JFrame
```

Figura 11.34 | Clases adaptadoras utilizadas para implementar los manejadores de eventos. (Parte 1 de 2).

```java
            } // fin de if
         } // fin del método mouseDragged
      } // fin de la clase interna anónima
   ); // fin de la llamada a addMouseMotionListener
} // fin del constructor de PanelDibujo

// dibuja un óvalo en un cuadro delimitador de 4 x 4, en la ubicación especificada en la ventana
public void paintComponent( Graphics g )
{
   super.paintComponent( g ); // borra el área de dibujo

   // dibuja todos los puntos en el arreglo
   for ( int i = 0; i < cuentaPuntos; i++ )
      g.fillOval( puntos[ i ].x, puntos[ i ].y, 4, 4 );
} // fin del método paint
} // fin de la clase PanelDibujo
```

Figura 11.34 | Clases adaptadoras utilizadas para implementar los manejadores de eventos. (Parte 2 de 2).

```java
// Fig. 11.35: Pintor.java
// Prueba de PanelDibujo.
import java.awt.BorderLayout;
import javax.swing.JFrame;
import javax.swing.JLabel;

public class Pintor
{
   public static void main( String args[] )
   {
      // crea objeto JFrame
      JFrame aplicacion = new JFrame( "Un programa simple de dibujo" );

      PanelDibujo panelDibujo = new PanelDibujo(); // crea panel de dibujo
      aplicacion.add( panelDibujo, BorderLayout.CENTER ); // en el centro

      // crea una etiqueta y la coloca en la región SOUTH de BorderLayout
      aplicacion.add( new JLabel( "Arrastre el raton para dibujar" ),
         BorderLayout.SOUTH );

      aplicacion.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
      aplicacion.setSize( 400, 200 ); // establece el tamaño del marco
      aplicacion.setVisible( true ); // muestra el marco
   } // fin de main
} // fin de la clase Pintor
```



Figura 11.35 | Clase de prueba de PanelDibujo.

En este ejemplo, utilizamos un arreglo de 10,000 objetos `Point` (línea 14) para almacenar la ubicación en la cual ocurre cada evento de arrastre del ratón. Como veremos más adelante, el método `paintComponent` utiliza estos objetos `Point` para dibujar. La variable de instancia `cuentaPuntos` (línea 11) mantiene el número total de objetos `Point` capturados de los eventos de arrastre del ratón hasta cierto punto.

Las líneas 20 a 35 registran un objeto `MouseMotionListener` para que escuche los eventos de movimiento del ratón de `PaintPanel`. Las líneas 22 a 34 crean un objeto de una clase interna anónima que extiende a la clase adaptadora `MouseMotionAdapter`. Recuerde que `MouseMotionAdapter` implementa a `MouseMotionListener`, por lo que el objeto de la clase interna anónima es un `MouseMotionListener`. La clase interna anónima hereda una implementación predeterminada de los métodos `mouseMoved` y `mouseDragged`, por lo que de antemano cumple con el requerimiento de que deben implementarse todos los métodos de la interfaz. Sin embargo, los métodos predeterminados no hacen nada cuando se les llama. Por lo tanto, sobrescribimos el método `mouseDragged` en las líneas 25 a 33 para capturar las coordenadas de un evento de arrastre del ratón y las almacenamos como un objeto `Point`. La línea 27 asegura que se almacenen las coordenadas del evento, sólo si aún hay elementos vacíos en el arreglo. De ser así, en la línea 29 se invoca el método **`getPoint`** de `MouseEvent` para obtener el objeto `Point` en donde ocurrió el evento, y lo almacena en el arreglo, en el índice `cuentaPuntos`. La línea 30 incrementa la `cuentaPuntos`, y la línea 31 llama al método **`repaint`** (heredado directamente de la clase `Component`) para indicar que el objeto `PanelDibujo` debe actualizarse en la pantalla lo más pronto posible, con una llamada al método `paintComponent` de `PaintPanel`.

El método `paintComponent` (líneas 39 a 46), que recibe un parámetro `Graphics`, se llama de manera automática cada vez que el objeto `PaintPanel` necesita mostrarse en la pantalla (como cuando se muestra por primera vez la GUI) o actualizarse en la pantalla (como cuando se hace una llamada al método `repaint`, o cuando otra ventana en la pantalla oculta el componente de la GUI y después se vuelve otra vez visible).



Observación de apariencia visual 11.14

Una llamada a `repaint` para un componente de GUI de Swing indica que el componente debe actualizarse en la pantalla lo más pronto que sea posible. El fondo del componente de GUI se borra sólo si el componente es opaco. El método `setOpaque` de `JComponent` puede recibir un argumento `boolean`, el cual indica si el componente es opaco (`true`) o transparente(`false`).

La línea 41 invoca a la versión de `paintComponent` de la superclase para borrar el fondo de `PanelDibujo` (los objetos `JPanel` son opacos de manera predeterminada). Las líneas 44 y 45 dibujan un óvalo en la ubicación especificada por cada objeto `Point` en el arreglo (hasta la `cuentaPuntos`). El método **`fillOval`** de `Graphics` dibuja un óvalo relleno. Los cuatro parámetros del método representan un área rectangular (que se conoce como **cuadro delimitador**) en la cual se muestra el óvalo. Los primeros dos parámetros son la coordenada *x* superior izquierda y la coordenada *y* superior izquierda del área rectangular. Las últimas dos coordenadas representan la anchura y la altura del área rectangular. El método `fillOval` dibuja el óvalo de manera que esté en contacto con la parte media de cada lado del área rectangular. En la línea 45, los primeros dos argumentos se especifican mediante el uso de las dos variables de instancia `public` de la clase `Point`: `x` y `y`. El ciclo termina cuando se encuentra una referencia `null` en el arreglo, o cuando se llega al final del mismo. En el capítulo 12 aprenderá más acerca de las características de `Graphics`.



Observación de apariencia visual 11.15

La acción de dibujar en cualquier componente de GUI se lleva a cabo con coordenadas que se miden a partir de la esquina superior izquierda (0, 0) de ese componente de la GUI, no de la esquina superior izquierda de la pantalla.

Uso del objeto `JPanel` personalizado en una aplicación

La clase `Pintor` (figura 11.35) contiene el método principal que ejecuta esta aplicación. En la línea 14 se crea un objeto `PanelDibujo`, en el cual el usuario puede arrastrar el ratón para dibujar. En la línea 15 se adjunta el objeto `PanelDibujo` al objeto `JFrame`.

11.16 Manejo de eventos de teclas

En esta sección presentamos la interfaz **`KeyListener`** para manejar **eventos de teclas**. Estos eventos se generan cuando se oprimen y sueltan las teclas en el teclado. Una clase que implementa a `KeyListener` debe proporcionar

declaraciones para los métodos **keyPressed**, **keyReleased** y **keyTyped**, cada uno de los cuales recibe un objeto **KeyEvent** como argumento. La clase KeyEvent es una subclase de InputEvent. El método keyPressed es llamado en respuesta a la acción de oprimir cualquier tecla. El método keyTyped es llamado en respuesta a la acción de oprimir una tecla que no sea una **tecla de acción**. (Las teclas de acción son cualquier tecla de dirección, *Inicio*, *Fin*, *Re Pág*, *Av Pág*, cualquier tecla de función, *Bloq Num*, *Impr Pant*, *Bloq Despl*, *Bloq Mayús* y *Pausa*). El método keyReleased es llamado cuando la tecla se suelta después de un evento keyPressed o keyTyped.

La aplicación de las figuras 11.36 y 11.37 demuestra el uso de los métodos de KeyListener. La clase DemoTeclas implementa la interfaz KeyListener, por lo que los tres métodos se declaran en la aplicación.

El constructor (figuras 11.36, líneas 17 a 28) registra a la aplicación para manejar sus propios eventos de teclas, utilizando el método **addKeyListener** en la línea 27. Este método se declara en la clase Component, por lo que todas las subclases de Component pueden notificar a objetos KeyListener acerca de los eventos para ese objeto Component.

```
1  // Fig. 11.36: MarcoDemoTeclas.java
2  // Demostración de los eventos de pulsación de teclas.
3  import java.awt.Color;
4  import java.awt.event.KeyListener;
5  import java.awt.event.KeyEvent;
6  import javax.swing.JFrame;
7  import javax.swing.JTextArea;
8
9  public class MarcoDemoTeclas extends JFrame implements KeyListener
10 {
11    private String linea1 = ""; // primera línea del área de texto
12    private String linea2 = ""; // segunda línea del área de texto
13    private String linea3 = ""; // tercera línea del área de texto
14    private JTextArea areaTexto; // área de texto para mostrar la salida
15
16    // constructor de MarcoDemoTeclas
17    public MarcoDemoTeclas()
18    {
19       super( "Demostración de los eventos de pulsacion de teclas" );
20
21       areaTexto = new JTextArea( 10, 15 ); // establece el objeto JTextArea
22       areaTexto.setText( "Oprima cualquier tecla en el teclado..." );
23       areaTexto.setEnabled( false ); // deshabilita el área de texto
24       areaTexto.setDisabledTextColor( Color.BLACK ); // establece el color del texto
25       add( areaTexto ); // agrega areaTexto a JFrame
26
27       addKeyListener( this ); // permite al marco procesar los eventos de teclas
28    } // fin del constructor de MarcoDemoTeclas
29
30    // maneja el evento de oprimir cualquier tecla
31    public void keyPressed( KeyEvent evento )
32    {
33       linea1 = String.format( "Tecla oprimida: %s",
34          evento.getKeyText( evento.getKeyCode() ) ); // imprime la tecla oprimida
35       establecerLineas2y3( evento ); // establece las líneas de salida dos y tres
36    } // fin del método keyPressed
37
38    // maneja el evento de liberar cualquier tecla
39    public void keyReleased( KeyEvent evento )
40    {
41       linea1 = String.format( "Tecla liberada: %s",
42          evento.getKeyText( evento.getKeyCode() ) ); // imprime la tecla liberada
43       establecerLineas2y3( evento ); // establece las líneas de salida dos y tres
```

Figura 11.36 | Manejo de eventos de teclas. (Parte 1 de 2).

```
    } // fin del método keyReleased

    // maneja el evento de oprimir una tecla de acción
    public void keyTyped( KeyEvent evento )
    {
       linea1 = String.format( "Tecla oprimida: %s", evento.getKeyChar() );
       establecerLineas2y3( evento ); // establece las líneas de salida dos y tres
    } // fin del método keyTyped

    // establece las líneas de salida dos y tres
    private void establecerLineas2y3( KeyEvent evento )
    {
       linea2 = String.format( "Esta tecla %s es una tecla de accion",
          ( evento.isActionKey() ? "" : "no " ) );

       String temp = evento.getKeyModifiersText( evento.getModifiers() );

       linea3 = String.format( "Teclas modificadoras oprimidas: %s",
          ( temp.equals( "" ) ? "ninguna" : temp ) ); // imprime modificadoras

       areaTexto.setText( String.format( "%s\n%s\n%s\n",
          linea1, linea2, linea3 ) ); // imprime tres líneas de texto
    } // fin del método establecerLineas2y3
 } // fin de la clase MarcoDemoTeclas
```

Figura 11.36 | Manejo de eventos de teclas. (Parte 2 de 2).

```
 // Fig. 11.37: DemoTeclas.java
 // Prueba de MarcoDemoTeclas.
 import javax.swing.JFrame;

 public class DemoTeclas
 {
    public static void main( String args[] )
    {
       MarcoDemoTeclas marcoDemoTeclas = new MarcoDemoTeclas();
       marcoDemoTeclas.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
       marcoDemoTeclas.setSize( 350, 100 ); // establece el tamaño del marco
       marcoDemoTeclas.setVisible( true ); // muestra el marco
    } // fin de main
 } // fin de la clase DemoTeclas
```

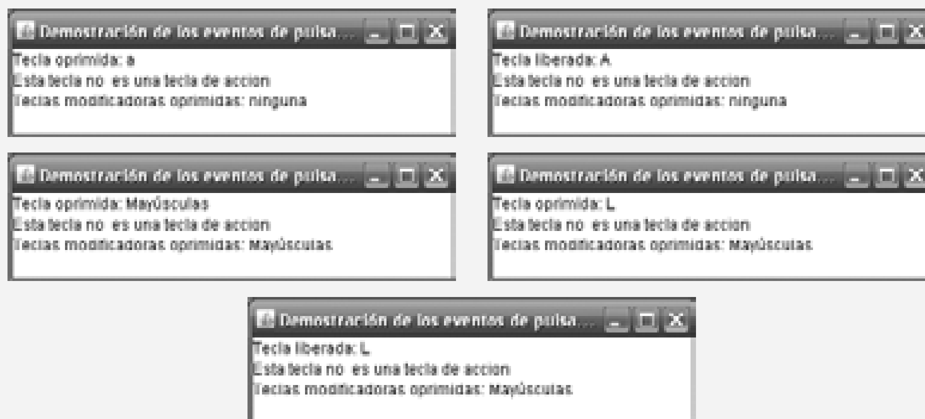


Figura 11.37 | Clase de prueba de `MarcoDemoTeclas`. (Parte 1 de 2).



Figura 11.37 | Clase de prueba de `MarcoDemoTeclas`. (Parte 2 de 2).

En la línea 25, el constructor agrega el objeto `JTextArea` llamado `areaTexto` (en donde se muestra la salida de la aplicación) al objeto `JFrame`. Observe en las capturas de pantalla que el objeto `areaTexto` ocupa toda la ventana. Esto se debe al esquema predeterminado `BorderLayout` del objeto `JFrame` (que describiremos en la sección 11.17.2 y demostraremos en la figura 11.41). Cuando se agrega un objeto `Component` individual a un objeto `BorderLayout`, el objeto `Component` ocupa todo el objeto `Container` completo. Observe que en la línea 24 se utiliza el método `setDisabledTextColor` para cambiar el color del texto en el área de texto a negro.

Los métodos `keyPressed` (líneas 31 a 36) y `keyReleased` (líneas 39 a 44) utilizan el método **`getKeyCode`** de `KeyEvent` para obtener el **código de tecla virtual** de la tecla oprimida. La clase `KeyEvent` mantiene un conjunto de constantes (las constantes de código de tecla virtual) que representa a todas las teclas en el teclado. Estas constantes pueden compararse con el valor de retorno de `getKeyCode` para probar teclas individuales en el teclado. El valor devuelto por `getKeyCode` se pasa al método **`getKeyText`** de `KeyEvent`, el cual devuelve una cadena que contiene el nombre de la tecla que se oprimió. Para obtener una lista completa de las constantes de teclas virtuales, vea la documentación en línea para la clase `KeyEvent` (paquete `java.awt.event`). El método `keyTyped` (líneas 47 a 51) utiliza el método **`getKeyChar`** de `KeyEvent` para obtener el valor Unicode del carácter escrito.

Los tres métodos manejadores de eventos terminan llamando al método `establecerLineas2y3` (líneas 54 a 66) y le pasan el objeto `KeyEvent`. Este método utiliza el método **`isActionKey`** de `KeyEvent` (línea 57) para determinar si la tecla en el evento fue una tecla de acción. Además, se hace una llamada al método **`getModifiers`** de `InputEvent` (línea 59) para determinar si se oprimió alguna tecla modificadora (como *Mayús*, *Alt* y *Ctrl*) cuando ocurrió el evento de tecla. El resultado de este método se pasa al método **`getKeyModifiersText`** de `KeyEvent`, el cual produce una cadena que contiene los nombres de las teclas modificadoras que se oprimieron.

[*Nota:* si necesita probar una tecla específica en el teclado, la clase `KeyEvent` proporciona una **constante de tecla** para cada tecla del teclado. Estas constantes pueden utilizarse desde los manejadores de eventos de teclas para determinar si se oprimió una tecla específica. Además, para determinar si las teclas *Alt*, *Ctrl*, *Meta* y *Mayús* se oprimen individualmente, cada uno de los métodos **`isAltDown`**, **`isControlDown`**, **`isMetaDown`** e **`isShiftDown`** devuelven un valor `boolean`, indicando si se oprimió dicha tecla durante el evento de tecla].

11.17 Administradores de esquemas

Los **administradores de esquemas** se proporcionan para ordenar los componentes de la GUI en un contenedor, para fines de presentación. Los programadores pueden usar los administradores de esquemas como herramientas básicas de distribución visual, en vez de determinar la posición y tamaño exactos de cada componente de la GUI. Esta funcionalidad permite al programador concentrarse en la vista y sentido básicos, y deja que el administrador de esquemas procese la mayoría de los detalles de la distribución visual. Todos los administradores de esquemas implementan la interfaz **`LayoutManager`** (en el paquete `java.awt`). El método `setLayout` de la clase `Container` toma un objeto que implementa a la interfaz `LayoutManager` como argumento. Básicamente, existen tres formas para poder ordenar los componentes en una GUI:

1. Posicionamiento absoluto: esto proporciona el mayor nivel de control sobre la apariencia de una GUI. Al establecer el esquema de un objeto `Container` en `null`, podemos especificar la posición absoluta de cada componente de la GUI con respecto a la esquina superior izquierda del objeto `Container`. Si hacemos esto, también debemos especificar el tamaño de cada componente de la GUI. La programación de una GUI con posicionamiento absoluto puede ser un proceso tedioso, a menos que se cuente con un entorno de desarrollo integrado (IDE), que pueda generar el código por nosotros.
2. Administradores de esquemas: el uso de administradores de esquemas para posicionar elementos puede ser un proceso más simple y rápido que la creación de una GUI con posicionamiento absoluto, pero se pierde cierto control sobre el tamaño y el posicionamiento preciso de los componentes de la GUI.

3. Programación visual en un IDE: los IDEs proporcionan herramientas que facilitan la creación de GUIs. Por lo general, cada IDE proporciona una **herramienta de diseño de GUI** que nos permite arrastrar y soltar componentes de GUI desde un cuadro de herramientas, hacia un área de diseño. Después podemos posicionar, ajustar el tamaño de los componentes de la GUI y alinearlos según lo deseado. El IDE genera el código de Java que crea la GUI. Además, podemos, por lo general, agregar código manejador de eventos para un componente específico, haciendo doble clic en el componente. Algunas herramientas de diseño también nos permiten utilizar los administradores de esquemas descritos en este capítulo y en el capítulo 22.



Observación de apariencia visual 11.16

La mayoría de los entornos de programación de Java proporcionan herramientas de diseño de la GUI, las cuales ayudan a un programador a diseñar gráficamente una GUI; después, las herramientas de diseño escriben código en Java para crear la GUI. Dichas herramientas proporcionan con frecuencia un mayor control sobre el tamaño, la posición y la alineación de los componentes de la GUI, en comparación con los administradores de esquemas integrados.



Observación de apariencia visual 11.17

Es posible establecer el esquema de un objeto `Container` en `null`, lo cual indica que no debe utilizarse ningún administrador de esquemas. En un objeto `Container` sin un administrador de esquemas, el programador debe posicionar y cambiar el tamaño de los componentes en el contenedor dado, y cuidar que, en los eventos de ajuste de tamaño, todos los componentes se reposicionen según sea necesario. Los eventos de ajuste de tamaño de un componente pueden procesarse mediante un objeto `ComponentListener`.

En la figura 11.38 se sintetizan los administradores de esquemas presentados en este capítulo. En el capítulo 22 hablaremos sobre otros administradores de esquemas.

Administrador de esquemas	Descripción
`FlowLayout`	Es el predeterminado para `javax.swing.JPanel`. Coloca los componentes secuencialmente (de izquierda a derecha) en el orden en que se agregaron. También es posible especificar el orden de los componentes utilizando el método `add` de `Container`, el cual toma un objeto `Component` y una posición de índice entero como argumentos.
`BorderLayout`	Es el predeterminado para los objetos `JFrame` (y otras ventanas). Ordena los componentes en cinco áreas: `NORTH`, `SOUTH`, `EAST`, `WEST` y `CENTER`.
`GridLayout`	Ordena los componentes en filas y columnas.

Figura 11.38 | Administradores de esquemas.

11.17.1 FlowLayout

Éste es el administrador de esquemas más simple. Los componentes de la GUI se colocan en un contenedor, de izquierda a derecha, en el orden en el que se agregaron al contenedor. Cuando se llega al borde del contenedor, los componentes siguen mostrándose en la siguiente línea. La clase `FlowLayout` permite a los componentes de la GUI **alinearse a la izquierda**, **al centro** (el valor predeterminado) y **a la derecha**.

La aplicación de las figuras 11.39 y 11.40 crea tres objetos `JButton` y los agrega a la aplicación, utilizando un administrador de esquemas `FlowLayout`. Los componentes se alinean hacia el centro de manera predeterminada. Cuando el usuario hace clic en **Izquierda**, la alineación del administrador de esquemas cambia a un `FlowLayout` alineado a la izquierda. Cuando el usuario hace clic en **Derecha**, la alineación del administrador de esquemas cambia a un `FlowLayout` alineado a la derecha. Cuando el usuario hace clic en **Centro**, la alineación del administrador de esquemas cambia a un `FlowLayout` alineado hacia el centro. Cada botón tiene su propio manejador

de eventos que se declara con una clase interna, la cual implementa a `ActionListener`. Las ventanas de salida de ejemplo muestran cada una de las alineaciones de `FlowLayout`. Además, la última ventana de salida de ejemplo muestra la alineación centrada después de ajustar el tamaño de la ventana a una anchura menor. Observe que el botón **Derecha** fluye hacia una nueva línea.

```
// Fig. 11.39: MarcoFlowLayout.java
// Demostración de las alineaciones de FlowLayout.
import java.awt.FlowLayout;
import java.awt.Container;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import javax.swing.JFrame;
import javax.swing.JButton;

public class MarcoFlowLayout extends JFrame
{
   private JButton botonJButtonIzquierda; // botón para establecer la alineación a la izquierda
   private JButton botonJButtonCentro; // botón para establecer la alineación al centro
   private JButton botonJButtonDerecha; // botón para establecer la alineación a la derecha
   private FlowLayout esquema; // objeto esquema
   private Container contenedor; // contenedor para establecer el esquema

   // establece la GUI y registra los componentes de escucha de botones
   public MarcoFlowLayout()
   {
      super( "Demostracion de FlowLayout" );

      esquema = new FlowLayout(); // crea objeto FlowLayout
      contenedor = getContentPane(); // obtiene contenedor para esquema
      setLayout( esquema ); // establece el esquema del marco

      // establece botonJButtonIzquierda y registra componente de escucha
      botonJButtonIzquierda = new JButton( "Izquierda" ); // crea botón Izquierda
      add( botonJButtonIzquierda ); // agrega botón Izquierda al marco
      botonJButtonIzquierda.addActionListener(

         new ActionListener() // clase interna anónima
         {
            // procesa evento de botonJButtonIzquierda
            public void actionPerformed( ActionEvent evento )
            {
               esquema.setAlignment( FlowLayout.LEFT );

               // realinea los componentes adjuntos
               esquema.layoutContainer( contenedor );
            } // fin del método actionPerformed
         } // fin de la clase interna anónima
      ); // fin de la llamada a addActionListener

      // establece botonJButtonCentro y registra componente de escucha
      botonJButtonCentro = new JButton( "Centro" ); // crea botón Centro
      add( botonJButtonCentro ); // agrega botón Centro al marco
      botonJButtonCentro.addActionListener(

         new ActionListener() // clase interna anónima
```

Figura 11.39 | `FlowLayout` permite a los componentes fluir a través de varias líneas. (Parte 1 de 2).

```
51            {
52               // procesa evento de botonJButtonCentro
53               public void actionPerformed( ActionEvent evento )
54               {
55                  esquema.setAlignment( FlowLayout.CENTER );
56
57                  // realinea los componentes adjuntos
58                  esquema.layoutContainer( contenedor );
59               } // fin del método actionPerformed
60            } // fin de la clase interna anónima
61         ); // fin de la llamada a addActionListener
62
63         // establece botonJButtonDerecha y registra componente de escucha
64         botonJButtonDerecha = new JButton( "Derecha" ); // crea botón Derecha
65         add( botonJButtonDerecha ); // agrega botón Derecha al marco
66         botonJButtonDerecha.addActionListener(
67
68            new ActionListener() // clase interna anónima
69            {
70               // procesa evento de botonJButtonDerecha
71               public void actionPerformed( ActionEvent evento )
72               {
73                  esquema.setAlignment( FlowLayout.RIGHT );
74
75                  // realinea los componentes adjuntos
76                  esquema.layoutContainer( contenedor );
77               } // fin del método actionPerformed
78            } // fin de la clase interna anónima
79         ); // fin de la llamada a addActionListener
80      } // fin del constructor de MarcoFlowLayout
81   } // fin de la clase MarcoFlowLayout
```

Figura 11.39 | `FlowLayout` permite a los componentes fluir a través de varias líneas. (Parte 2 de 2).

Como se vio anteriormente, el esquema de un contenedor se establece mediante el método `setLayout` de la clase `Container`. En la línea 25 se establece el administrador de esquemas en `FlowLayout`, el cual se declara en la línea 23. Generalmente, el esquema se establece antes de agregar cualquier componente de la GUI a un contenedor.

```
// Fig. 11.40: DemoFlowLayout.java
// Prueba de MarcoFlowLayout.
import javax.swing.JFrame;

public class DemoFlowLayout
{
   public static void main( String args[] )
   {
      MarcoFlowLayout marcoFlowLayout = new MarcoFlowLayout();
      marcoFlowLayout.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
      marcoFlowLayout.setSize( 350, 75 ); // establece el tamaño del marco
      marcoFlowLayout.setVisible( true ); // muestra el marco
   } // fin de main
} // fin de la clase DemoFlowLayout
```

Figura 11.40 | Clase de prueba de `MarcoFlowLayout`. (Parte 1 de 2).

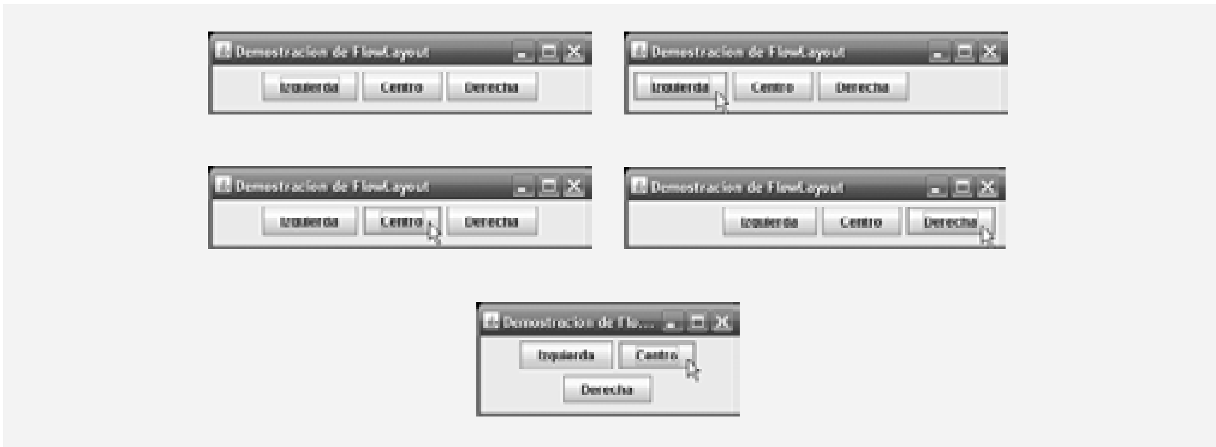


Figura 11.40 | Clase de prueba de `MarcoFlowLayout`. (Parte 2 de 2).



Observación de apariencia visual 11.18

Cada contenedor puede tener solamente un administrador de esquemas. Varios contenedores separados en el mismo programa pueden tener distintos administradores de esquemas.

Observe en este ejemplo que el manejador de eventos de cada botón se especifica con un objeto de una clase interna anónima separada (líneas 30 a 43, 48 a 61 y 66 a 71, respectivamente). El manejador de eventos `actionPerformed` de cada botón ejecuta dos instrucciones. Por ejemplo, la línea 37 en el método `actionPerformed` para el botón `botonJButtonIzquierda` utiliza el método **`setAlignment`** de `FlowLayout` para cambiar la alineación del objeto `FlowLayout` a la izquierda (**`FlowLayout.LEFT`**). En la línea 40 se utiliza el método **`layoutContainer`** de la interfaz `LayoutManager` (que todos los administradores de esquemas heredan) para especificar que el objeto `JFrame` debe reordenarse, con base en el esquema ajustado. Dependiendo del botón oprimido, el método `actionPerformed` para cada botón establece la alineación del objeto `FlowLayout` a `FlowLayout.LEFT` (línea 37), **`FlowLayout.CENTER`** (línea 55) o **`FlowLayout.RIGHT`** (línea 73).

11.17.2 BorderLayout

El administrador de esquemas `BorderLayout` (el predeterminado para un objeto `JFrame`) ordena los componentes en cinco regiones: `NORTH`, `SOUTH`, `EAST`, `WEST` y `CENTER`. `NORTH` corresponde a la parte superior del contenedor. La clase `BorderLayout` extiende a `Object` e implementa a la interfaz **`LayoutManager2`** (una subinterfaz de `LayoutManager`, que agrega varios métodos para un mejor procesamiento de los esquemas).

Un `BorderLayout` limita a un objeto `Container` para que contenga cuando mucho cinco componentes; uno en cada región. El componente que se coloca en cada región puede ser un contenedor, al cual se pueden adjuntar otros componentes. Los componentes que se colocan en las regiones `NORTH` y `SOUTH` se extienden horizontalmente hacia los lados del contenedor, y tienen la misma altura que los componentes que se colocan en esas regiones. Las regiones `EAST` y `WEST` se expanden verticalmente entre las regiones `NORTH` y `SOUTH`, y tienen la misma anchura que los componentes que se coloquen dentro de ellas. El componente que se coloca en la región `CENTER` se expande para rellenar todo el espacio restante en el esquema (esto explica por qué el objeto `JTextArea` de la figura 11.36 ocupa toda la ventana). Si las cinco regiones están ocupadas, todo el espacio del contenedor se cubre con los componentes de la GUI. Si las regiones `NORTH` o `SOUTH` no están ocupadas, los componentes de la GUI en las regiones `EAST`, `CENTER` y `WEST` se expanden verticalmente para rellenar el espacio restante. Si las regiones `EAST` o `WEST` no están ocupadas, el componente de la GUI en la región `CENTER` se expande horizontalmente para rellenar el espacio restante. Si la región `CENTER` no está ocupada, el área se deja vacía; los demás componentes de la GUI no se expanden para rellenar el espacio restante. La aplicación de las figuras 11.41 y 11.42 demuestra el administrador de esquemas `BorderLayout`, utilizando cinco objetos `JButton`.

En la línea 21 se crea un objeto `BorderLayout`. Los argumentos del constructor especifican el número de píxeles entre los componentes que se ordenan en forma horizontal (**espacio libre horizontal**) y el número

de píxeles entre los componentes que se ordenan en forma vertical (**espacio libre vertical**), respectivamente. El valor predeterminado es un píxel de espacio libre horizontal y vertical. En la línea 22 se utiliza el método `setLayout` para establecer el esquema del panel de contenido en `esquema`.

```
1  // Fig. 11.41: MarcoBorderLayout.java
2  // Demostración de BorderLayout.
3  import java.awt.BorderLayout;
4  import java.awt.event.ActionListener;
5  import java.awt.event.ActionEvent;
6  import javax.swing.JFrame;
7  import javax.swing.JButton;
8
9  public class MarcoBorderLayout extends JFrame implements ActionListener
10 {
11    private JButton botones[]; // arreglo de botones para ocultar porciones
12    private final String nombres[] = { "Ocultar Norte", "Ocultar Sur",
13       "Ocultar Este", "Ocultar Oeste", "Ocultar Centro" };
14    private BorderLayout esquema; // objeto BorderLayout
15
16    // establece la GUI y el manejo de eventos
17    public MarcoBorderLayout()
18    {
19       super( "Demostracion de BorderLayout" );
20
21       esquema = new BorderLayout( 5, 5 ); // espacios de 5 píxeles
22       setLayout( esquema ); // establece el esquema del marco
23       botones = new JButton[ nombres.length ]; // establece el tamaño del arreglo
24
25       // crea objetos JButton y registra componentes de escucha para ellos
26       for ( int cuenta = 0; cuenta < nombres.length; cuenta++ )
27       {
28          botones[ cuenta ] = new JButton( nombres[ cuenta ] );
29          botones[ cuenta ].addActionListener( this );
30       } // fin de for
31
32       add( botones[ 0 ], BorderLayout.NORTH ); // agrega botón al norte
33       add( botones[ 1 ], BorderLayout.SOUTH ); // agrega botón al sur
34       add( botones[ 2 ], BorderLayout.EAST ); // agrega botón al este
35       add( botones[ 3 ], BorderLayout.WEST ); // agrega botón al oeste
36       add( botones[ 4 ], BorderLayout.CENTER ); // agrega botón al centro
37    } // fin del constructor de MarcoBorderLayout
38
39    // maneja los eventos de botón
40    public void actionPerformed( ActionEvent evento )
41    {
42       // comprueba el origen del evento y distribuye el panel de contenido de manera
         acorde
43       for ( JButton boton : botones )
44       {
45          if ( evento.getSource() == boton )
46             boton.setVisible( false ); // oculta el botón oprimido
47          else
48             boton.setVisible( true ); // muestra los demás botones
49       } // fin de for
50
51       esquema.layoutContainer( getContentPane() ); // distribuye el panel de contenido
52    } // fin del método actionPerformed
53 } // fin de la clase MarcoBorderLayout
```

Figura 11.41 | `BorderLayout` que contiene cinco botones.

```java
// Fig. 11.42: DemoBorderLayout.java
// Prueba de MarcoBorderLayout.
import javax.swing.JFrame;

public class DemoBorderLayout
{
   public static void main( String args[] )
   {
      MarcoBorderLayout marcoBorderLayout = new MarcoBorderLayout();
      marcoBorderLayout.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
      marcoBorderLayout.setSize( 375, 200 ); // establece el tamaño del marco
      marcoBorderLayout.setVisible( true ); // muestra el marco
   } // fin de main
} // fin de la clase DemoBorderLayout
```



Figura 11.42 | Clase de prueba de `MarcoBorderLayout`.

Agregamos objetos `Component` a un objeto `BorderLayout` con otra versión del método `add` de `Container` que toma dos argumentos: el objeto `Component` que se va a agregar y la región en la que debe aparecer este objeto. Por ejemplo, en la línea 32 se especifica que `botones[ 0 ]` debe aparecer en la región `NORTH`. Los componentes pueden agregarse en cualquier orden, pero sólo debe agregarse un componente a cada región.



Observación de apariencia visual 11.19

Si no se especifica una región al agregar un objeto `Component` *a un objeto* `BorderLayout`*, el administrador de esquemas asume que el objeto* `Component` *debe agregarse a la región* `BorderLayout.CENTER`*.*



Error común de programación 11.6

Cuando se agrega más de un componente a una región en un objeto `BorderLayout`*, sólo se mostrará el último componente agregado a esa región. No hay un error que indique este problema.*

Observe que la clase `MarcoBorderLayout` implementa directamente a `ActionListener` en este ejemplo, por lo que el objeto `MarcoBorderLayout` manejará los eventos de los objetos `JButton`. Por esta razón, en la línea 29 se pasa la referencia `this` al método `addActionListener` de cada objeto `JButton`. Cuando el usuario hace clic en un objeto `JButton` específico en el esquema, se ejecuta el método `actionPerformed` (líneas 40 a 52). La instrucción `for` mejorada en las líneas 43 a 49 utiliza una instrucción `if...else` para ocultar el objeto `JButton` específico que generó el evento. El método **`setVisible`** (que `JButton` hereda de la clase `Component`) se llama con un argumento `false` (línea 46) para ocultar el objeto `JButton`. Si el objeto `JButton` actual en el arreglo no es el que generó el evento, se hace una llamada al método `setVisible` con un argumento `true` (línea 48) para asegurar que el objeto `JButton` se muestre en la pantalla. En la línea 51 se utiliza el método `layoutContainer` de `LayoutManager` para recalcular la distribución visual del panel de contenido. Observe en las capturas de pantalla de la figura 11.41 que ciertas regiones en el objeto `BorderLayout` cambian de forma a medida que se ocultan objetos `JButton` y se muestran en otras regiones. Pruebe a cambiar el tamaño de la ventana de la aplicación para ver cómo las diversas regiones ajustan su tamaño, con base en la anchura y la altura de la ventana. Para esquemas más complejos, agrupe los componentes en objetos `JPanel`, cada uno con un administrador de esquemas separado. Coloque los objetos `JPanel` en el objeto `JFrame`, usando el esquema `BorderLayout` predeterminado o cualquier otro esquema.

11.17.3 GridLayout

El administrador de esquemas **`GridLayout`** divide el contenedor en una cuadrícula, de manera que los componentes puedan colocarse en filas y columnas. La clase `GridLayout` hereda directamente de la clase `Object` e implementa a la interfaz `LayoutManager`. Todo objeto `Component` en un objeto `GridLayout` tiene la misma anchura y altura. Los componentes se agregan a un objeto `GridLayout` empezando en la celda superior izquierda de la cuadrícula, y procediendo de izquierda a derecha hasta que la fila esté llena. Después el proceso continúa de izquierda a derecha en la siguiente fila de la cuadrícula, y así sucesivamente. La aplicación de las figuras 11.43 y 11.44 demuestra el administrador de esquemas `GridLayout`, utilizando seis objetos `JButton`.

```java
// Fig. 11.43: MarcoGridLayout.java
// Demostración de GridLayout.
import java.awt.GridLayout;
import java.awt.Container;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import javax.swing.JFrame;
import javax.swing.JButton;

public class MarcoGridLayout extends JFrame implements ActionListener
{
   private JButton botones[]; // arreglo de botones
   private final String nombres[] =
      { "uno", "dos", "tres", "cuatro", "cinco", "seis" };
   private boolean alternar = true; // alterna entre dos esquemas
   private Container contenedor; // contenedor del marco
   private GridLayout cuadricula1; // primer objeto GridLayout
   private GridLayout cuadricula2; // segundo objeto GridLayout

   // constructor sin argumentos
   public MarcoGridLayout()
   {
      super( "Demostracion de GridLayout" );
```

Figura 11.43 | `GridLayout` que contiene seis botones. (Parte 1 de 2).

```
        cuadricula1 = new GridLayout( 2, 3, 5, 5 ); // 2 por 3; espacios de 5
        cuadricula2 = new GridLayout( 3, 2 ); // 3 por 2; sin espacios
        contenedor = getContentPane(); // obtiene el panel de contenido
        setLayout( cuadricula1 ); // establece esquema de objeto JFrame
        botones = new JButton[ nombres.length ]; // crea arreglo de objetos JButton

        for ( int cuenta = 0; cuenta < nombres.length; cuenta++ )
        {
           botones[ cuenta ] = new JButton( nombres[ cuenta ] );
           botones[ cuenta ].addActionListener( this ); // registra componente de escucha
           add( botones[ cuenta ] ); // agrega boton a objeto JFrame
        } // fin de for
     } // fin del constructor de MarcoGridLayout

     // maneja eventos de boton, alternando entre los esquemas
     public void actionPerformed( ActionEvent evento )
     {
        if ( alternar )
           contenedor.setLayout( cuadricula2 ); // establece esquema al primero
        else
           contenedor.setLayout( cuadricula1 ); // establece esquema al segundo

        alternar = !alternar; // establece alternar a su valor opuesto
        contenedor.validate(); // redistribuye el contenedor
     } // fin del método actionPerformed
  } // fin de la clase MarcoGridLayout
```

Figura 11.43 | `GridLayout` que contiene seis botones. (Parte 2 de 2).

```
// Fig. 11.44: DemoGridLayout.java
// Prueba de MarcoGridLayout.
import javax.swing.JFrame;

public class DemoGridLayout
{
   public static void main( String args[] )
   {
      MarcoGridLayout marcoGridLayout = new MarcoGridLayout();
      marcoGridLayout.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
      marcoGridLayout.setSize( 300, 200 ); // establece el tamaño del marco
      marcoGridLayout.setVisible( true ); // muestra el marco
   } // fin de main
} // fin de la clase DemoGridLayout
```





Figura 11.44 | Clase de prueba de `MarcoGridLayout`.

En las líneas 24 y 25 se crean dos objetos `GridLayout`. El constructor de `GridLayout` que se utiliza en la línea 24 especifica un objeto `GridLayout` con 2 filas, 3 columnas, 5 píxeles de espacio libre horizontal entre objetos `Component` en la cuadrícula y 5 píxeles de espacio libre vertical entre objetos `Component` en la cuadrícula. El constructor de `GridLayout` que se utiliza en la línea 25 especifica un objeto `GridLayout` con 3 filas y 2 columnas que utiliza el espacio libre predeterminado (1 píxel).

Los objetos `JButton` en este ejemplo se ordenan inicialmente utilizando `cuadricula1` (que se establece para el panel de contenido en la línea 27, mediante el método `setLayout`). El primer componente se agrega a la primera columna de la primera fila. El siguiente componente se agrega a la segunda columna de la primera fila, y así sucesivamente. Cuando se oprime un objeto `JButton`, se hace una llamada al método `actionPerformed` (líneas 39 a 48). Todas las llamadas a `actionPerformed` alternan el esquema entre `cuadricula2` y `cuadricula1`, utilizando la variable `boolean` llamada `alternar` para determinar el siguiente esquema a establecer.

En la línea 47 se muestra otra manera para cambiar el formato a un contenedor para el cual haya cambiado el esquema. El método **`validate`** de `Container` recalcula el esquema del contenedor, con base en el administrador de esquemas actual para ese objeto `Container` y el conjunto actual de componentes de la GUI que se muestran en pantalla.

11.18 Uso de paneles para administrar esquemas más complejos

Las GUIs complejas (como la de la figura 11.1) requieren que cada componente se coloque en una ubicación exacta. A menudo consisten de varios paneles, en donde los componentes de cada panel se ordenan en un esquema específico. La clase `JPanel` extiende a `JComponent`, y `JComponent` extiende a la clase `Container`, por lo que todo `JPanel` es un `Container`. Por lo tanto, todo objeto `JPanel` puede tener componentes, incluyendo otros paneles, los cuales se adjuntan mediante el método `add` de `Container`. La aplicación de las figuras 11.45 y 11.46 demuestra cómo puede usarse un objeto `JPanel` para crear un esquema más complejo, en el cual se coloquen varios objetos `JButton` en la región `SOUTH` de un esquema `BorderLayout`.

Una vez declarado el objeto `JPanel` llamado `panelBotones` en la línea 11, y creado en la línea 19, en la línea 20 se establece el esquema de `panelBotones` en `GridLayout` con una fila y cinco columnas (hay cinco objetos `JButton` en el arreglo `botones`). En las líneas 23 a 27 se agregan los cinco objetos `JButton` del arreglo `botones` al objeto `JPanel` en el ciclo. En la línea 26 se agregan los botones directamente al objeto `JPanel` (la clase `JPanel` no tiene un panel de contenido, a diferencia de `JFrame`). En la línea 29 se utiliza el objeto `BorderLayout` predeterminado para agregar `panelBotones` a la región `SOUTH`. Observe que esta región tiene la misma altura que los botones en `panelBotones`. Un objeto `JPanel` ajusta su tamaño de acuerdo con los componentes que contiene. A medida que se agregan más componentes, el objeto `JPanel` crece (de acuerdo con las restricciones de su administrador de esquemas) para dar cabida a esos nuevos componentes. Ajuste el tamaño de la ventana para que vea cómo el administrador de esquemas afecta al tamaño de los objetos `JButton`.

```
1  // Fig. 11.45: MarcoPanel.java
2  // Uso de un objeto JPanel para ayudar a distribuir los componentes.
3  import java.awt.GridLayout;
4  import java.awt.BorderLayout;
5  import javax.swing.JFrame;
6  import javax.swing.JPanel;
7  import javax.swing.JButton;
8  
9  public class MarcoPanel extends JFrame
10 {
11    private JPanel panelBotones; // panel que contiene los botones
12    private JButton botones[]; // arreglo de botones
13 
14    // constructor sin argumentos
15    public MarcoPanel()
16    {
```

Figura 11.45 | `Jpanel` con cinco objetos `JButton`, en un esquema `GridLayout` adjunto a la región `SOUTH` de un esquema `BorderLayout`. (Parte 1 de 2).

```
      super( "Demostracion de Panel" );
      botones = new JButton[ 5 ]; // crea el arreglo botones
      panelBotones = new JPanel(); // establece el panel
      panelBotones.setLayout( new GridLayout( 1, botones.length ) );

      // crea y agrega los botones
      for ( int cuenta = 0; cuenta < botones.length; cuenta++ )
      {
         botones[ cuenta ] = new JButton( "Boton " + ( cuenta + 1 ) );
         panelBotones.add( botones[ cuenta ] ); // agrega el botón al panel
      } // fin de for

      add( panelBotones, BorderLayout.SOUTH ); // agrega el panel a JFrame
   } // fin del constructor de MarcoPanel
} // fin de la clase MarcoPanel
```

Figura 11.45 | `Jpanel` con cinco objetos `JButton`, en un esquema `GridLayout` adjunto a la región `SOUTH` de un esquema `BorderLayout`. (Parte 2 de 2).

```
// Fig. 11.46: DemoPanel.java
// Prueba de MarcoPanel.
import javax.swing.JFrame;

public class DemoPanel extends JFrame
{
   public static void main( String args[] )
   {
      MarcoPanel marcoPanel = new MarcoPanel();
      marcoPanel.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
      marcoPanel.setSize( 450, 200 ); // establece el tamaño del marco
      marcoPanel.setVisible( true ); // muestra el marco
   } // fin de main
} // fin de la clase DemoPanel
```



Figura 11.46 | Clase de prueba de `MarcoPanel`.

11.19 JTextArea

Un objeto **`JTextArea`** proporciona un área para manipular varias líneas de texto. Al igual que la clase `JTextField`, `JTextArea` es una subclase de `JTextComponent`, el cual declara métodos comunes para objetos `JTextField`, `JTextArea` y varios otros componentes de GUI basados en texto.

La aplicación en las figuras 11.47 y 11.48 demuestra el uso de los objetos `JTextArea`. Un objeto `JTextArea` muestra texto que el usuario puede seleccionar. El otro objeto `JTextArea` no puede editarse, y se utiliza para mostrar el texto que seleccionó el usuario en el primer objeto `JTextArea`. A diferencia de los objetos `JTextField`, los objetos `JTextArea` no tienen eventos de acción. Al igual que con los objetos `JList` de selección múltiple (sección

11.12), un evento externo de otro componente de GUI indica cuándo se debe procesar el texto en un objeto `JTextArea`. Por ejemplo, al escribir un mensaje de correo electrónico, por lo general, hacemos clic en un botón **Enviar** para enviar el texto del mensaje al recipiente. De manera similar, al editar un documento en un procesador de palabras, por lo general, guardamos el archivo seleccionando un elemento de menú llamado **Guardar** o **Guardar como…**. En este programa, el botón **Copiar >>>** genera el evento externo que copia el texto seleccionado en el objeto `JTextArea` de la izquierda, y lo muestra en el objeto `JTextArea` de la derecha.

```java
// Fig. 11.47: MarcoAreaTexto.java
// Copia el texto seleccionado de un área de texto a otra.
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import javax.swing.Box;
import javax.swing.JFrame;
import javax.swing.JTextArea;
import javax.swing.JButton;
import javax.swing.JScrollPane;

public class MarcoAreaTexto extends JFrame
{
   private JTextArea areaTexto1; // muestra cadena de demostración
   private JTextArea areaTexto2; // el texto resaltado se copia aquí
   private JButton botonCopiar; // inicia el copiado de texto

   // constructor sin argumentos
   public MarcoAreaTexto()
   {
      super( "Demostracion de JTextArea" );
      Box cuadro = Box.createHorizontalBox(); // crea un cuadro
      String demo = "Esta es una cadena de\ndemostracion para\n" +
         "ilustrar como copiar texto\nde un area de texto a \n" +
         "otra, usando un\nevento externo\n";

      areaTexto1 = new JTextArea( demo, 10, 15 ); // crea área de texto 1
      cuadro.add( new JScrollPane( areaTexto1 ) ); // agrega panel de desplazamiento

      botonCopiar = new JButton( "Copiar >>>" ); // crea botón para copiar
      cuadro.add( botonCopiar ); // agrega botón de copia al cuadro
      botonCopiar.addActionListener(

         new ActionListener() // clase interna anónima
         {
            // establece el texto en areaTexto2 con el texto seleccionado de areaTexto1
            public void actionPerformed( ActionEvent evento )
            {
               areaTexto2.setText( areaTexto1.getSelectedText() );
            } // fin del método actionPerformed
         } // fin de la clase interna anónima
      ); // fin de la llamada a addActionListener

      areaTexto2 = new JTextArea( 10, 15 ); // crea segunda área de texto
      areaTexto2.setEditable( false ); // deshabilita edición
      cuadro.add( new JScrollPane( areaTexto2 ) ); // agrega panel de desplazamiento

      add( cuadro ); // agrega cuadro al marco
   } // fin del constructor de MarcoAreaTexto
} // fin de la clase MarcoAreaTexto
```

Figura 11.47 | Copiado de texto seleccionado, de un objeto `JTextArea` a otro.

```
// Fig. 11.48: DemoAreaTexto.java
// Copia el texto seleccionado de un área de texto a otra.
import javax.swing.JFrame;

public class DemoAreaTexto
{
   public static void main( String args[] )
   {
      MarcoAreaTexto marcoAreaTexto = new MarcoAreaTexto();
      marcoAreaTexto.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
      marcoAreaTexto.setSize( 425, 200 ); // establece el tamaño del marco
      marcoAreaTexto.setVisible( true ); // muestra el marco
   } // fin de main
} // fin de la clase DemoAreaTexto
```

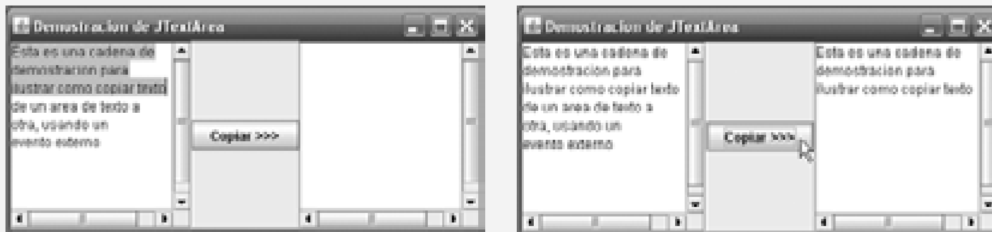


Figura 11.48 | Copiado de texto seleccionado, de un objeto `TextAreaFrame`.

En el constructor (líneas 18 a 48), la línea 21 crea un contenedor **`Box`** (paquete `javax.swing`) para organizar los componentes de la GUI. `Box` es una subclase de `Container` que utiliza un administrador de esquemas **`BoxLayout`** (que veremos con detalle en la sección 22.9) para ordenar los componentes de la GUI, ya sea en forma horizontal o vertical. El método `static` **`createHorizontalBox`** de `Box` crea un objeto `Box` que ordena los componentes de izquierda a derecha, en el orden en el que se adjuntan.

En las líneas 26 y 43 se crean los objetos `JTextArea` llamados `areaTexto1` y `areaTexto2`. La línea 26 utiliza el constructor con tres argumentos de `JTextArea`, el cual recibe un objeto `String` que representa el texto inicial y dos valores `int` que especifican que el objeto `JTextArea` tiene `10` filas y `15` columnas. En la línea 43 se utiliza el constructor con dos argumentos de `JTextArea`, el cual especifica que el objeto `JTextArea` tiene `10` filas y `15` columnas. En la línea 26 se especifica que `demo` debe mostrarse como el contenido predeterminado del objeto `JTextArea`. Un objeto `JTextArea` no proporciona barras de desplazamiento si no puede mostrar su contenido completo. Por lo tanto, en la línea 27 se crea un objeto **`JScrollPane`**, se inicializa con `areaTexto1` y se adjunta al contenedor `cuadro`. En un objeto `JScrollPane` aparecen de manera predeterminada las barras de desplazamiento horizontal y vertical, según sea necesario.

En las líneas 29 a 41 se crea el objeto `JButton` llamado `botonCopiar` con la etiqueta `"Copiar >>>"`, se agrega `botonCopiar` al contenedor `cuadro` y se registra el manejador de eventos para el evento `ActionEvent` de `botonCopiar`. Este botón proporciona el evento externo que determina cuándo debe copiar el programa el texto seleccionado en `areaTexto1` a `areaTexto2`. Cuando el usuario hace clic en `botonCopiar`, la línea 38 en `actionPerformed` indica que el método **`getSelectedText`** (que hereda `JTextArea` de `JTextComponent`) debe devolver el texto seleccionado de `areaTexto1`. Para seleccionar el texto, el usuario arrastra el ratón sobre el texto deseado para resaltarlo. El método `setText` cambia el texto en `areaTexto2` por la cadena que devuelve `getSelectedText`.

En las líneas 43 a 45 se crea `areaTexto2`, se establece su propiedad editable a `false` y se agrega al contenedor `box`. En la línea 47 se agrega `cuadro` al objeto `JFrame`. En la sección 11.17 vimos que el esquema predeterminado de un objeto `JFrame` es `BorderLayout`, y que el método `add` adjunta de manera predeterminada su argumento a la región `CENTER` de este esquema.

Algunas veces es conveniente, cuando el texto llega al lado derecho de un objeto `JTextArea`, hacer que se recorra a la siguiente línea. A esto se le conoce como **envoltura de línea**. La clase `JTextArea` no envuelve líneas de manera predeterminada.



Observación de apariencia visual 11.20

Para proporcionar la funcionalidad de envoltura de líneas para un objeto `JTextArea`*, invoque el método* ***setLineWrap*** *de* `JTextArea` *con un argumento* `true`*.*

Políticas de las barras de desplazamiento de `JScrollPane`

En este ejemplo se utiliza un objeto `JScrollPane` para proporcionar la capacidad de desplazamiento a un objeto `JTextArea`. De manera predeterminada, `JScrollPane` muestra las barras de desplazamiento sólo si se requieren. Puede establecer las **políticas de las barras de desplazamiento** horizontal y vertical de un objeto `JScrollPane` al momento de crearlo. Si un programa tiene una referencia a un objeto `JScrollPane`, puede usar los métodos **`setHorizontalScrollBarPolicy`** y **`setVerticalScrollBarPolicy`** de `JScrollPane` para modificar las políticas de las barras redesplazamiento en cualquier momento. La clase `JScrollPane` declara las constantes

```
JScrollPane.VERTICAL_SCROLLBAR_ALWAYS
JScrollPane.HORIZONTAL_SCROLLBAR_ALWAYS
```

para indicar que siempre debe aparecer una barra de desplazamiento, las constantes

```
JScrollPane.VERTICAL_SCROLLBAR_AS_NEEDED
JScrollPane.HORIZONTAL_SCROLLBAR_AS_NEEDED
```

para indicar que debe aparecer una barra de desplazamiento sólo si es necesario (los valores predeterminados), y las constantes

```
JScrollPane.VERTICAL_SCROLLBAR_NEVER
JScrollPane.HORIZONTAL_SCROLLBAR_NEVER
```

para indicar que nunca debe aparecer una barra de desplazamiento. Si la política de la barra de desplazamiento horizontal se establece en `JScrollPane.HORIZONTAL_SCROLLBAR_NEVER`, un objeto `JTextArea` adjunto al objeto `JScrollPane` envolverá las líneas de manera automática.

11.20 Conclusión

En este capítulo aprendió acerca de muchos componentes de la GUI, y cómo implementar el manejo de eventos. También aprendió acerca de las clases anidadas, las clases internas y las clases internas anónimas. Vio la relación especial entre un objeto de la clase interna y un objeto de su clase de nivel superior. Aprendió a utilizar diálogos `JOptionPane` para obtener datos de entrada de texto del usuario, y cómo mostrar mensajes a éste. También aprendió a crear aplicaciones que se ejecuten en sus propias ventanas. Hablamos sobre la clase `JFrame` y los componentes que permiten a un usuario interactuar con una aplicación. También aprendió cómo mostrar texto e imágenes al usuario. Vimos cómo personalizar los objetos `JPanel` para crear áreas de dibujo personalizadas, las cuales utilizará ampliamente en el siguiente capítulo. Vio cómo organizar los componentes en una ventana mediante el uso de los administradores de esquemas, y cómo crear GUIs más complejas mediante el uso de objetos `JPanel` para organizar los componentes. Por último, aprendió acerca del componente `JTextArea`, en el cual un usuario puede introducir texto y una aplicación puede mostrarlo. En el capítulo 22, Componentes de la GUI: parte 2, aprenderá acerca de los componentes de GUI más avanzados, como los botones deslizables, los menús y los administradores de esquemas más complicados. En el siguiente capítulo aprenderá a agregar gráficos a su aplicación de GUI. Los gráficos nos permiten dibujar figuras y texto con colores y estilos.

Resumen

Sección 11.1 Introducción

- Una interfaz gráfica de usuario (GUI) presenta un mecanismo amigable al usuario para interactuar con una aplicación. Una GUI proporciona a una aplicación una “apariencia visual” única.
- Al proporcionar distintas aplicaciones en las que los componentes de la interfaz de usuario sean consistentes e intuitivos, los usuarios pueden familiarizarse en cierto modo con una aplicación, de manera que pueden aprender a utilizarla en menor tiempo y con mayor productividad.

- Las GUIs se crean a partir de componentes de GUI; a éstos se les conoce algunas veces como controles o "widgets".

Sección 11.2 Entrada/salida simple basada en GUI con `JOptionPane`

- La mayoría de las aplicaciones utilizan ventanas o cuadros de diálogo (también conocidos como diálogos) para interactuar con el usuario.
- La clase `JOptionPane` de Java (paquete `javax.swing`) proporciona cuadros de diálogo preempaquetados para entrada y salida. El método `static showInputDialog` de `JOptionPane` muestra un diálogo de entrada.
- Por lo general, un indicador utiliza la capitalización estilo oración: un estilo que capitaliza sólo la primera letra de la primera palabra en el texto, a menos que la palabra sea un nombre propio.
- Un diálogo de entrada sólo puede introducir objetos `String`. Esto es común en la mayoría de los componentes de la GUI.
- El método `static showMessageDialog` de `JOptionPane` muestra un diálogo de mensaje.

Sección 11.3 Generalidades de los componentes de Swing

- La mayoría de los componentes de GUI de Swing se encuentran en el paquete `javax.swing.` Forman parte de las Java Foundation Classes (JFC): las bibliotecas de Java para el desarrollo de GUIs en distintas plataformas.
- En conjunto, a la apariencia y la forma en la que interactúa el usuario con la aplicación se les denomina la apariencia visual. Los componentes de GUI de Swing nos permiten especificar una apariencia visual uniforme para una aplicación a través de todas las plataformas, o para usar la apariencia visual personalizada de cada plataforma.
- Los componentes ligeros de Swing no están enlazados a los componentes actuales de GUI que soporte la plataforma subyacente en la que se ejecuta una aplicación.
- Varios componentes de Swing son componentes pesados, que requieren una interacción directa con el sistema de ventanas local, lo cual puede restringir su apariencia y funcionalidad.
- La clase `Component` (paquete `java.awt`) declara muchos de los atributos y comportamientos comunes para los componentes de GUI en los paquetes `java.awt` y `javax.swing`.
- La clase `Container` (paquete `java.awt`) es una subclase de `Component`. Los objetos `Component` se adjuntan a los objetos `Container`, de manera que puedan organizarse y mostrarse en la pantalla.
- La clase `JComponent` (paquete `javax.swing`) es una subclase de `Container. JComponent` es la superclase de todos los componentes ligeros de Swing, y declara los atributos y comportamientos comunes.
- Algunas de las características comunes de `JComponent` son: una apariencia visual adaptable, teclas de método abreviado llamadas nemónicos, cuadros de información sobre herramientas, soporte para tecnologías de ayuda y soporte para la localización de la interfaz de usuario.

Sección 11.4 Mostrar texto e imágenes en una ventana

- La mayoría de las ventanas son instancias de la clase `JFrame` o una subclase de `JFrame`. `JFrame` proporciona los atributos y comportamientos básicos de una ventana.
- Un objeto `JLabel` muestra una sola línea de texto de sólo lectura, una imagen, o texto y una imagen. Por lo general, el texto en un objeto `JLabel` usa la capitalización estilo oración.
- Al crear una GUI, cada componente de ésta debe adjuntarse a un contenedor, como una ventana creada con un objeto `JFrame`.
- Muchos IDEs proporcionan herramientas de diseño de GUIs, en las cuales podemos especificar el tamaño y la ubicación exactos de un componente mediante el uso del ratón, y después el IDE genera el código de la GUI por nosotros.
- El método `setToolTipText` de `JComponent` especifica la información sobre herramientas que se muestra cuando el usuario coloca el cursor del ratón sobre un componente ligero.
- El método `add` de `Container` adjunta un componente de GUI a un objeto `Container`.
- La clase `ImageIcon` (paquete `javax.swing`) soporta varios formatos de imagen, incluyendo GIF, PNG y JPEG.
- El método `getClass` (de la clase `Object`) obtiene una referencia al objeto `Class` que representa la declaración de la clase para el objeto en el que se hace la llamada al método.
- El método `getResource` de `Class` devuelve la ubicación de su argumento en forma de URL. El método `getResource` usa el cargador de clases del objeto `Class` para determinar la ubicación del recurso.
- La interfaz `SwingConstants` (paquete `javax.swing`) declara un conjunto de constantes enteras comunes que se utilizan con muchos componentes de Swing.
- Las alineaciones horizontal y vertical de un objeto `JLabel` se pueden establecer mediante los métodos `setHorizontalAlignment` y `setVerticalAlignment`, respectivamente.
- El método `setText` de `JLabel` establece el texto a mostrar en una etiqueta. El correspondiente método `getText` obtiene el texto actual que se muestra en una etiqueta.

- El método `setIcon` de `JLabel` especifica el objeto `Icon` a mostrar en una etiqueta. El correspondiente método `getIcon` obtiene el objeto `Icon` actual que se muestra en una etiqueta.
- Los métodos `setHorizontalTextPosition` y `setVerticalTextPosition` de `JLabel` especifican la posición del texto en la etiqueta.
- El método `setDefaultCloseOperation` de `JFrame`, con la constante `JFrame.EXIT_ON_CLOSE` como argumento, indica que el programa debe terminar cuando el usuario cierre la ventana.
- El método `setSize` de `Component` especifica la anchura y la altura de un componente.
- El método `setVisible` de `Component` con el argumento `true` muestra un objeto `JFrame` en la pantalla.

Sección 11.5 Campos de texto y una introducción al manejo de eventos con clases anidadas

- Las GUIs se controlan por eventos; cuando el usuario interactúa con un componente de GUI, los eventos controlan al programa para realizar las tareas.
- El código que realiza una tarea en respuesta a un evento se llama manejador de eventos, y el proceso general de responder a los eventos se conoce como manejo de eventos.
- La clase `JTextField` extiende a la clase `JTextComponent` (paquete `javax.swing.text`), que proporciona muchas características comunes para los componentes de Swing basados en texto. La clase `JPasswordField` extiende a `JTextField` y agrega varios métodos específicos para el procesamiento de contraseñas.
- Un objeto `JPasswordField` muestra que se están escribiendo caracteres a medida que el usuario los introduce, pero oculta los caracteres reales con caracteres de eco.
- Un componente recibe el enfoque cuando el usuario hace clic sobre él.
- El método `setEditable` de `JTextComponent` puede usarse para hacer que un campo de texto no pueda editarse.
- Antes de que una aplicación pueda responder a un evento para un componente específico de la GUI, debemos realizar varios pasos de codificación: 1) Crear una clase que represente al manejador de eventos. 2) Implementar una interfaz apropiada, conocida como interfaz de escucha de eventos, en la clase del *paso 1*. 3) Indicar que se debe notificar a un objeto de la clase de los *pasos 1* y 2 cuando ocurra el evento. A esto se le conoce como registrar el manejador de eventos.
- Las clases anidadas pueden ser `static` o no `static`. Las clases anidadas no `static` se llaman clases internas, y se utilizan con frecuencia para el manejo de eventos.
- Antes de poder crear un objeto de una clase interna, debe haber primero un objeto de la clase de nivel superior, que contenga a la clase interna, ya que un objeto de la clase interna tiene de manera implícita una referencia a un objeto de su clase de nivel superior.
- Un objeto de la clase interna puede acceder directamente a todas las variables de instancia y métodos de su clase de nivel superior.
- Una clase anidada que sea `static` no requiere un objeto de su clase de nivel superior, y no tiene de manera implícita una referencia a un objeto de la clase de nivel superior.
- Cuando el usuario oprime *Intro* en un objeto `JTextField` o `JPasswordField`, el componente de la GUI genera un evento `ActionEvent` (paquete `java.awt.event`). Dicho evento se procesa mediante un objeto que implementa a la interfaz `ActionListener` (paquete `java.awt.event`).
- El método `addActionListener` de `JTextField` registra el manejador de eventos para un campo de texto de un componente. Este método recibe como argumento un objeto `ActionListener`.
- El componente de GUI con el que interactúa el usuario es el `origen del evento`.
- Un objeto `ActionEvent` contiene información acerca del evento que acaba de ocurrir, como el origen del evento y el texto en el campo de texto.
- El método `getSource` de `ActionEvent` devuelve una referencia al origen del evento. El método `getActionCommand` de `ActionEvent` devuelve el texto que escribió el usuario en un campo de texto o en la etiqueta de un objeto `JButton`.
- El método `getPassword` de `JPasswordField` devuelve la contraseña que escribió el usuario.

Sección 11.6 Tipos de eventos comunes de la GUI e interfaces de escucha

- Para cada tipo de objeto evento hay, por lo general, una interfaz de escucha de eventos que le corresponde. Cada interfaz de escucha de eventos especifica uno o más métodos manejadores de eventos, que deben declararse en la clase que implementa a la interfaz.

Sección 11.7 Cómo funciona el manejo de eventos

- Cuando ocurre un evento, el componente de la GUI con el que el usuario interactuó notifica a sus componentes de escucha registrados, llamando al método de manejo de eventos apropiado de cada componente de escucha.

- Todo objeto `JComponent` tiene una variable de instancia llamada `listenerList`, la cual hace referencia a un objeto de la clase `EventListenerList` (paquete `javax.swing.event`). Cada objeto de una subclase de `JComponent` mantiene las referencias a todos sus componentes de escucha registrados en la variable `listenerList`.
- Todo componente de la GUI soporta varios tipos de eventos, incluyendo los eventos de ratón, de teclado y otros. Cuando ocurre un evento, éste se despacha sólo a los componentes de escucha de eventos del tipo apropiado. El componente de la GUI recibe un ID de evento único, especificando el tipo de evento, el cual utiliza para decidir el tipo de componente de escucha al que debe despacharse el evento, y cuál método llamar en cada objeto componente de escucha.

Sección 11.8 `JButton`

- Un botón es un componente en el que el usuario hace clic para desencadenar cierta acción. Todos los tipos de botones son subclases de `AbstractButton` (paquete `javax.swing`), la cual declara las características comunes para los botones de Swing. Por lo general, las etiquetas de los botones usan la capitalización tipo título de libro; un estilo que capitaliza la primera letra de cada palabra significativa en el texto, y no termina con ningún signo de puntuación.
- Los botones de comandos se crean con la clase `JButton`.
- Un objeto `JButton` puede mostrar un objeto `Icon`. Para proporcionar al usuario un nivel adicional de interacción visual con la GUI, un objeto `JButton` también puede tener un icono de sustitución: un objeto `Icon` que se muestra cuando el usuario coloca el ratón sobre el botón.
- El método `setRolloverIcon` (de la clase `AbstractButton`) especifica la imagen a mostrar en un botón, cuando el usuario coloca el ratón sobre él.

Sección 11.9 Botones que mantienen el estado

- Los componentes de la GUI de Swing contienen tres tipos de botones de estado: `JToggleButton`, `JCheckBox` y `JRadioButton`.
- Las clases `JCheckBox` y `JRadioButton` son subclases de `JToggleButton`. Un objeto `JRadioButton` es distinto de un objeto `JCheckBox` en cuanto a que, generalmente, hay varios objetos `JRadioButton` que se agrupan, y sólo puede seleccionarse un botón en el grupo, en un momento dado.
- El método `setFont` (de la clase `Component`) establece el tipo de letra de un componente a un nuevo objeto de la clase `Font` (paquete `java.awt`).
- Cuando el usuario hace clic en un objeto `JCheckBox`, ocurre un evento `ItemEvent`. Este evento puede manejarse mediante un objeto `ItemListener`, que debe implementar al método `itemStateChanged`. El método `addItemListener` registra el componente de escucha para un objeto `JCheckBox` o `JRadioButton`.
- El método `isSelected` de `JCheckBox` determina si un objeto `JCheckBox` está seleccionado.
- Los objetos `JRadioButton` son similares a los objetos `JCheckBox` en cuanto a que tienen dos estados: seleccionado y no seleccionado. Sin embargo, generalmente los botones de opción aparecen como un grupo, en el cual sólo puede seleccionarse un botón a la vez. Al seleccionar un botón de opción distinto, se obliga a los demás botones de opción a deseleccionarse.
- Los objetos `JRadioButton` se utilizan para representar opciones mutuamente exclusivas.
- La relación lógica entre los objetos `JRadioButton` se mantiene mediante un objeto `ButtonGroup` (paquete `javax.swing`).
- El método `add` de `ButtonGroup` asocia a cada objeto `JRadioButton` con un objeto `ButtonGroup`. Si se agrega más de un objeto `JRadioButton` seleccionado a un grupo, el primer objeto `JRadioButton` seleccionado que se agregue será el que quede seleccionado cuando se muestre la GUI en pantalla.
- Los objetos `JRadioButton` generan eventos `ItemEvent` cuando se hace clic sobre ellos.

Sección 11.10 `JComboBox` *y el uso de una clase interna anónima para el manejo de eventos*

- Un objeto `JComboBox` proporciona una lista de elementos, de los cuales el usuario puede seleccionar uno. Los objetos `JComboBox` generan eventos `ItemEvent`.
- Cada elemento en un objeto `JComboBox` tiene un índice. El primer elemento que se agrega a un objeto `JComboBox` aparece como el elemento actualmente seleccionado cuando se muestra el objeto `JComboBox`. Los otros elementos se seleccionan haciendo clic en el objeto `JComboBox`, el cual se expande en una lista, de la cual el usuario puede seleccionar un elemento.
- El método `setMaximumRowCount` de `JComboBox` establece el máximo número de elementos a mostrar cuando el usuario haga clic en el objeto `JComboBox`. Si hay elementos adicionales, el objeto `JComboBox` proporciona una barra de desplazamiento que permite al usuario desplazarse por todos los elementos en la lista.
- Una clase interna anónima es una forma especial de clase interna, que se declara sin un nombre y por lo general aparece dentro de la declaración de un método. Como una clase interna anónima no tiene nombre, un objeto de la clase interna anónima debe crearse en el punto en el que se declara la clase.
- El método `getSelectedIndex` de `JcomboBox` devuelve el índice del elemento seleccionado.

Sección 11.11 `JList`

- Un objeto `JList` muestra una serie de elementos, de los cuales el usuario puede seleccionar uno o más. La clase `JList` soporta las listas de selección simple y de selección múltiple.
- Cuando el usuario hace clic en un elemento de un objeto `JList`, se produce un evento `ListSelectionEvent`. El método `addListSelectionListener` registra un objeto `ListSelectionListener` para los eventos de selección de un objeto `JList`. Un objeto `ListSelectionListener` (paquete `javax.swing.event`) debe implementar el método `valueChanged`.
- El método `setVisibleRowCount` de `JList` especifica el número de elementos visibles en la lista.
- El método `setSelectionMode` de `JList` especifica el modo de selección de una lista.
- Un objeto `JList` no proporciona una barra de desplazamiento si hay más elementos en la lista que el número de filas visibles. En este caso, puede usarse un objeto `JScrollPane` para proporcionar la capacidad de desplazamiento. El método `getContentPane` de `JFrame` devuelve una referencia al panel de contenido de `JFrame`, en donde se muestran los componentes de la GUI.
- El método `getSelectedIndex` de `JList` devuelve el índice del elemento seleccionado.

Sección 11.12 Listas de selección múltiple

- Una lista de selección múltiple permite al usuario seleccionar muchos elementos de un objeto `JList`.
- El método `setFixedCellWidth` de `JList` establece la anchura de un objeto `JList`. El método `setFixedCellHeight` establece la altura de cada elemento en un objeto `JList`.
- No hay eventos para indicar que un usuario ha realizado varias selecciones en una lista de selección múltiple. Por lo general, un evento externo generado por otro componente de la GUI especifica cuándo deben procesarse las selecciones múltiples en un objeto `JList`.
- El método `setListData` de `JList` establece los elementos a mostrar en un objeto `JList`. El método `getSelectedValues` de `JList` devuelve un arreglo de objetos `Object` que representan los elementos seleccionados en un objeto `JList`.

Sección 11.13 Manejo de eventos del ratón

- Las interfaces de escucha de eventos `MouseListener` y `MouseMotionListener` se utilizan para manejar los eventos del ratón. Estos eventos se pueden atrapar para cualquier componente de la GUI que extienda a `Component`.
- La interfaz `MouseInputListener` (paquete `javax.swing.event`) extiende a las interfaces `MouseListener` y `MouseMotionListener` para crear una sola interfaz que contenga a todos sus métodos.
- Cada uno de los métodos manejadores de eventos del ratón recibe un objeto `MouseEvent` como argumento. Un objeto `MouseEvent` contiene información acerca del evento de ratón que ocurrió, incluyendo las coordenadas *x* y *y* de la ubicación en donde ocurrió el evento. Estas coordenadas se miden empezando desde la esquina superior izquierda del componente de la GUI en donde ocurrió el evento.
- Los métodos y constantes de la clase `InputEvent` (superclase de `MouseEvent`) permiten a una aplicación determinar cuál botón oprimió el usuario.
- La interfaz `MouseWheelListener` permite a las aplicaciones responder a la rotación de la rueda de un ratón.
- Los componentes de la GUI heredan los métodos `addMouseListener` y `addMouseMotionListener` de la clase `Component`.

Sección 11.14 Clases adaptadoras

- Muchas interfaces de escucha de eventos contienen varios métodos. Para muchas de estas interfaces, los paquetes `java.awt.event` y `javax.swing.event` proporcionan clases adaptadoras de escucha de eventos. Una clase adaptadora implementa a una interfaz y proporciona una implementación predeterminada de cada método en la interfaz. Podemos extender una clase adaptadora para que herede la implementación predeterminada de cada método, y por consiguiente, podemos sobrescribir sólo el (los) método(s) necesario(s) para el manejo de eventos.
- El método `getClickCount` de `MouseEvent` devuelve el número de clics de los botones del ratón. Los métodos `isMetaDown` e `isAltDown` determinan cuál botón del ratón oprimió el usuario.

Sección 11.15 Subclase de `JPanel` *para dibujar con el ratón*

- Los componentes ligeros de Swing que extienden a la clase `JComponent` contienen el método `paintComponent`, el cual se llama cuando se muestra un componente ligero de Swing. Al sobrescribir este método, puede especificar cómo dibujar figuras usando las herramientas de gráficos de Java.
- Al personalizar un objeto `JPanel` para usarlo como un área dedicada de dibujo, la subclase debe sobrescribir el método `paintComponent` y llamar a la versión de `paintComponent` de la superclase como la primera instrucción en el cuerpo del método sobrescrito.

- Las subclases de JComponent soportan la transparencia. Cuando un componente es opaco, paintComponent borra el fondo del componente antes de mostrarlo en pantalla.
- La transparencia de un componente ligero de Swing puede establecerse con el método setOpaque (un argumento false indica que el componente es transparente).
- La clase Point (paquete java.awt) representa una coordenada *x-y*.
- La clase Graphics se utiliza para dibujar.
- El método getPoint de MouseEvent obtiene el objeto Point en donde ocurrió un evento de ratón.
- El método repaint (heredado directamente de la clase Component) indica que un componente debe actualizarse en la pantalla lo más pronto posible.
- El método paintComponent recibe un parámetro Graphics, y se llama de manera automática cada vez que un componente ligero necesita mostrarse en la pantalla.
- El método fillOval de Graphics dibuja un óvalo relleno. Los cuatro parámetros del método representan el cuadro delimitador en el cual se muestra el óvalo. Los primeros dos parámetros son la coordenada *x* superior izquierda y la coordenada *y* superior izquierda del área rectangular. Las últimas dos coordenadas representan la anchura y la altura del área rectangular.

Sección 11.16 Manejo de eventos de teclas

- La interfaz KeyListener se utiliza para manejar eventos de teclas, que se generan cuando se oprimen y sueltan las teclas en el teclado. El método addKeyListener de la clase Component registra un objeto KeyListener para un componente.
- El método getKeyCode de KeyEvent obtiene el código de tecla virtual de la tecla oprimida. La clase KeyEvent mantiene un conjunto de constantes de código de tecla virtual que representa a todas las teclas en el teclado.
- El método getKeyText de KeyEvent devuelve una cadena que contiene el nombre de la tecla que se oprimió.
- El método getKeyChar de KeyEvent obtiene el valor Unicode del carácter escrito.
- El método isActionKey de KeyEvent determina si la tecla en un evento fue una tecla de acción.
- El método getModifiers de InputEvent determina si se oprimió alguna tecla modificadora (como *Mayús*, *Alt* y *Ctrl*) cuando ocurrió el evento de tecla.
- El método getKeyModifiersText de KeyEvent produce una cadena que contiene los nombres de las teclas modificadoras que se oprimieron.

Sección 11.17 Administradores de esquemas

- Los administradores de esquemas ordenan los componentes de la GUI en un contenedor, para fines de presentación.
- Todos los administradores de esquemas implementan la interfaz LayoutManager (paquete java.awt).
- El método setLayout de la clase Container especifica el esquema de un contenedor.
- FlowLayout es el administrador de esquemas más simple. Los componentes de la GUI se colocan en un contenedor, de izquierda a derecha, en el orden en el que se agregaron al contenedor. Cuando se llega al borde del contenedor, los componentes siguen mostrándose en la siguiente línea. La clase FlowLayout permite a los componentes de la GUI alinearse a la izquierda, al centro (el valor predeterminado) y a la derecha.
- El método setAlignment de FlowLayout cambia la alineación para un objeto FlowLayout.
- El administrador de esquemas BorderLayout (el predeterminado para un objeto JFrame) ordena los componentes en cinco regiones: NORTH, SOUTH, EAST, WEST y CENTER. NORTH corresponde a la parte superior del contenedor.
- Un BorderLayout limita a un objeto Container para que contenga cuando mucho cinco componentes; uno en cada región.
- El administrador de esquemas GridLayout divide el contenedor en una cuadrícula, de manera que los componentes puedan colocarse en filas y columnas.
- El método validate de Container recalcula el esquema del contenedor, con base en el administrador de esquemas actual para ese objeto Container y el conjunto actual de componentes de la GUI que se muestran en pantalla.

Sección 11.19 **JTextArea**

- Un objeto JTextArea proporciona un área para manipular varias líneas de texto. JTextArea es una subclase de JTextComponent, la cual declara métodos comunes para objetos JTextField, JTextArea y varios otros componentes de GUI basados en texto.
- La clase Box es una subclase de Container que utiliza un administrador de esquemas BoxLayout para ordenar los componentes de la GUI, ya sea en forma horizontal o vertical.

- El método `static createHorizontalBox` de `Box` crea un objeto `Box` que ordena los componentes de izquierda a derecha, en el orden en el que se adjuntan.
- El método `getSelectedText` (que hereda `JTextArea` de `JTextComponent`) devuelve el texto seleccionado de un objeto `JTextArea`.
- Podemos establecer las políticas de las barras de desplazamiento horizontal y vertical de un objeto `JScrollPane` al momento de crearlo. Los métodos `setHorizontalScrollBarPolicy` y `setVerticalScrollBarPolicy` de `JScrollPane` pueden usarse para modificar las políticas de las barras de desplazamiento en cualquier momento.

Terminología

`AbstractButton`, clase
`ActionEvent`, clase
`ActionListener`, interfaz
`actionPerformed`, método de `ActionListener`
`add`, método de `Container`
`add`, método de la clase `ButtonGroup`
`addActionListener`, método de la clase `JTextField`
`addItemListener`, método de la clase `AbstractButton`
`addKeyListener`, método de la clase `Component`
`addListSelectionListener`, método de la clase `JList`
`addMouseListener`, método de la clase `Component`
`addMouseMotionListener`, método de la clase `Component`
`addWindowListener`, método de la clase `JFrame`
administrador de esquemas
apariencia visual
área dedicada de dibujo
`AWTEvent`, clase
`BorderLayout`, clase
`Box`, clase
`BoxLayout`, clase
`ButtonGroup`, clase
capitalización tipo título de libro
clase adaptadora
clase adaptadora de escucha de eventos
clase anidada
clase de nivel superior
clase interna
clase interna anónima
clase `static` anidada
`Class`, clase
`Component`, clase
componente de escucha de eventos
componente de GUI
componente de GUI ligero
componente de GUI pesado
componentes de GUI de Swing
constructor predeterminado de una clase interna anónima
`Container`, clase
controlado por eventos
`createHorizontalBox`, método de la clase `Box`
cuadro de diálogo
despachar un evento
diálogo de entrada
diálogo de mensaje
enfoque
escribir en un campo de texto
`EventListenerList`, clase
evento
`fillOval`, método de la clase `Graphics`
`FlowLayout`, clase
`Font`, clase
`getActionCommand`, método de `ActionEvent`
`getClass`, método de `Object`
`getClickCount`, método de `MouseEvent`
`getContentPane`, método de `JFrame`
`getIcon`, método de `JLabel`
`getKeyChar`, método de `KeyEvent`
`getKeyCode`, método de `KeyEvent`
`getKeyModifiersText`, método de `KeyEvent`
`getKeyText`, método de `KeyEvent`
`getModifiers`, método de `InputEvent`
`getPassword`, método de `JPasswordField`
`getPoint`, método de `MouseEvent`
`getResource`, método de `Class`
`getSelectedIndex`, método de `JComboBox`
`getSelectedIndex`, método de `JList`
`getSelectedText`, método de `JTextComponent`
`getSelectedValues`, método de `JList`
`getSource`, método de `EventObject`
`getStateChange`, método de `ItemEvent`
`getText`, método de `JLabel`
`getX`, método de `MouseEvent`
`getY`, método de `MouseEvent`
`Graphics`, clase
`GridLayout`, clase
`Icon`, interfaz
icono de sustitución
`ImageIcon`, clase
información sobre herramientas
`InputEvent`, clase
interfaz de escucha de eventos
interfaz gráfica de usuario (GUI)
`isActionKey`, método de `KeyEvent`
`isAltDown`, método de `InputEvent`
`isAltDown`, método de `MouseEvent`
`isControlDown`, método de `InputEvent`

isMetaDown, método de InputEvent
isMetaDown, método de MouseEvent
isSelected, método de JCheckBox
isShiftDown, método de InputEvent
ItemEvent, clase
ItemListener, interfaz
itemStateChanged, método de ItemListener
java.awt, paquete
java.awt.event, paquete
javax.swing, paquete
javax.swing.event, paquete
JButton, clase
JCheckBox, clase
JComboBox, clase
JComponent, clase
JFrame, clase
JLabel, clase
JList, clase
JOptionPane, clase
JPanel, clase
JPasswordField, clase
JRadioButton, clase
JScrollPane, clase
JSlider, clase
JTextArea, clase
JTextComponent, clase
JTextField, clase
JToggleButton, clase
KeyAdapter, clase
KeyEvent, clase
KeyListener, interfaz
keyPressed, método de KeyListener
keyReleased, método de KeyListener
keyTyped, método de KeyListener
layoutContainer, método de LayoutManager
LayoutManager, interfaz
LayoutManager2, interfaz
listenerList, campo de JComponent
ListSelectionEvent, clase
ListSelectionListener, interfaz
ListSelectionModel, clase
manejador de eventos
manejo de eventos
modelo de eventos por delegación
MouseAdapter, clase
mouseClicked, método de MouseListener
mouseDragged, método de MouseMotionListener
mouseEntered, método de MouseListener
MouseEvent, clase
mouseExited, método de MouseListener
MouseInputListener, interfaz
MouseListener, interfaz
MouseMotionAdapter, clase
MouseMotionListener, interfaz
mouseMoved, método de MouseMotionListener
mousePressed, método de MouseListener
mouseReleased, método de MouseListener
MouseWheelEvent, clase
MouseWheelListener, interfaz
mouseWheelMoved, método de MouseWheelListener
objeto evento
origen del evento
paintComponent, método de JComponent
panel de contenido
Point, clase
registro de un evento
registro de un manejador de eventos
repaint, método de Component
setAlignment, método de FlowLayout
setBackground, método de Component
setDefaultCloseOperation, método de JFrame
setEditable, método de JTextComponent
setFixedCellHeight, método de JList
setFixedCellWidth, método de JList
setFont, método de Component
setHorizontalAlignment, método de JLabel
setHorizontalScrollBarPolicy, método de JScroll-Pane
setHorizontalTextPosition, método de JLabel
setIcon, método de JLabel
setLayout, método de Container
setLineWrap, método de JTextArea
setListData, método de JList
setMaximumRowCount, método de JComboBox
setOpaque, método de JComponent
setRolloverIcon, método de AbstractButton
setSelectionMode, método de JList
setSize, método de JFrame
setText, método de JLabel
setText, método de JTextComponent
setToolTipText, método de JComponent
setVerticalAlignment, método de JLabel
setVerticalScrollBarPolicy, método de JSlider
setVerticalTextPosition, método de JLabel
setVisible, método de Component
setVisible, método de JFrame
setVisibleRowCount, método de JList
showInputDialog, método de JOptionPane
showMessageDialog, método de JOptionPane
SwingConstants, interfaz
transparencia de un objeto JComponent
validate, método de Container
valueChanged, método de ListSelectionListener
WindowAdapter, clase
windowClosing, método de WindowListener
WindowListener, interfaz

Ejercicios de autoevaluación

11.1 Complete las siguientes oraciones:

a) El método ____________ es llamado cuando el ratón se mueve sin oprimir los botones y un componente de escucha de eventos está registrado para manejar el evento.
b) El texto que no puede ser modificado por el usuario se llama texto ____________.
c) Un ____________ ordena los componentes de la GUI en un objeto `Container`.
d) El método `add` para adjuntar componentes de la GUI es un método de la clase ____________.
e) GUI es un acrónimo para ____________.
f) El método ____________ se utiliza para especificar el administrador de esquemas para un contenedor.
g) Una llamada al método `mouseDragged` va precedida por una llamada al método ____________ y va seguida de una llamada al método ____________.
h) La clase ____________ contiene métodos que muestran diálogos de mensaje y diálogos de entrada.
i) Un diálogo de entrada capaz de recibir entrada del usuario se muestra con el método ____________ de la clase ____________.
j) Un diálogo capaz de mostrar un mensaje al usuario se muestra con el método ____________ de la clase ____________.
k) JTextField y `JTextArea` extienden a la clase ____________.

11.2 Conteste con *verdadero* o *falso* a cada una de las siguientes proposiciones; en caso de ser *falso*, explique por qué.

a) `BorderLayout` es el administrador de esquemas predeterminado para un panel de contenido de `JFrame`.
b) Cuando el cursor del ratón se mueve hacia los límites de un componente de la GUI, se hace una llamada al método `mouseOver`.
c) Un objeto `JPanel` no puede agregarse a otro `JPanel`.
d) En un esquema `BorderLayout`, dos botones que se agreguen a la región `NORTH` se mostrarán uno al lado del otro.
e) Cuando se utiliza `BorderLayout`, sólo deben mostrarse un máximo de cinco componentes.
f) Las clases internas no pueden acceder a los miembros de la clase que las encierra.
g) El texto de un objeto `JTextArea` siempre es de sólo lectura.
h) La clase `JTextArea` es una subclase directa de la clase `Component`.

11.3 Encuentre el (los) error(es) en cada una de las siguientes instrucciones y explique cómo corregirlo(s).

a)
```
nombreBoton = JButton( "Leyenda" );
```
b)
```
JLabel unaEtiqueta, JLabel;   // crear referencias
```
c)
```
campoTexto = new JTextField( 50, "Texto predeterminado" );
```
d)
```
Container contenedor = getContentPane();
setLayout( new BorderLayout() );
boton1 = new JButton( "Estrella del norte" );
boton2 = new JButton( "Polo sur" );
contenedor.add( boton1 );
contenedor.add( boton2 );
```

Respuestas a los ejercicios de autoevaluación

11.1 a) `mouseMoved`. b) no editable (de sólo lectura). c) administrador de esquemas. d) `Container`. e) interfaz gráfica de usuario. f) `setLayout`. g) `mousePressed`, `mouseReleased`. h) `JOptionPane`. i) `showInputDialog`, `JOptionPane`. j) `showMessageDialog`, `JOptionPane`. k) `JTextComponent`.

11.2

a) Verdadero.
b) Falso. Se hace una llamada al método `mouseEntered`.
c) Falso. Un `JPanel` puede agregarse a otro `JPanel`, ya que `JPanel` es una subclase indirecta de `Component`. Por lo tanto, un `JPanel` es un `Component`. Cualquier `Component` puede agregarse a un `Container`.
d) Falso. Sólo se mostrará el último botón que se agregue. Recuerde que sólo debe agregarse un componente a cada región en un esquema `BorderLayout`.
e) Verdadero.
f) Falso. Las clases internas tienen acceso a todos los miembros de la declaración de la clase que las encierra.

g) Falso. Los objetos `JTextArea` pueden editarse de manera predeterminada.
h) Falso. `JTextArea` se deriva de la clase `JTextComponent`.

11.3 a) Se necesita `new` para crear un objeto.
b) `JLabel` es el nombre de una clase y no puede utilizarse como nombre de variable.
c) Los argumentos que se pasan al constructor están invertidos. El objeto `String` debe pasarse primero.
d) Se ha establecido `BorderLayout` y los componentes se agregarán sin especificar la región, por lo que ambos se agregarán a la región central. Las instrucciones `add` apropiadas serían:

```
contenedor.add( boton1, BorderLayout.NORTH );
contenedor.add( boton2, BorderLayout.SOUTH );
```

Ejercicios

11.4 Complete las siguientes oraciones:
a) La clase `JTextField` extiende directamente a la clase ____________.
b) El método ____________ de `Container` adjunta un componente de la GUI a un contenedor.
c) El método ____________ es llamado cuando se suelta uno de los botones del ratón (sin mover el ratón).
d) La clase ____________ se utiliza para crear un grupo de objetos `JRadioButton`.

11.5 Conteste con *verdadero* o *falso* a cada una de las siguientes proposiciones; en caso de ser *falso*, explique por qué.
a) Sólo puede usarse un administrador de esquemas por cada objeto `Container`.
b) Los componentes de la GUI pueden agregarse a un objeto `Container` en cualquier orden, en un esquema `BorderLayout`.
c) Los objetos `JRadioButton` proporcionan una serie de opciones mutuamente exclusivas (es decir, sólo uno puede ser `true` en un momento dado).
d) El método `setFont` de `Graphics` se utiliza para establecer el tipo de letra para los campos de texto.
e) Un objeto `JList` muestra una barra de desplazamiento si hay más elementos en la lista de los que puedan mostrarse en pantalla.
f) Un objeto `Mouse` tiene un método llamado `mouseDragged`.

11.6 Conteste con *verdadero* o *falso* a cada una de las siguientes proposiciones; en caso de ser *falso*, explique por qué.
a) Un objeto `JPanel` es un objeto `JComponent`.
b) Un objeto `JPanel` es un objeto `Component`.
c) Un objeto `JLabel` es un objeto `Container`.
d) Un objeto `JList` es un objeto `JPanel`.
e) Un objeto `AbstractButton` es un objeto `JButton`.
f) Un objeto `JTextField` es un objeto `Object`.
g) `ButtonGroup` es una subclase de `JComponent`.

11.7 Encuentre los errores en cada una de las siguientes líneas de código y explique cómo corregirlos.
a) `import javax.swing.JFrame`
b) `objetoPanel.GridLayout( 8, 8 );  // establecer esquema GridLayout`
c) `contenedor.setLayout( new FlowLayout( FlowLayout.DEFAULT ) );`
d) `contenedor.add( botonEste, EAST );  // BorderLayout`

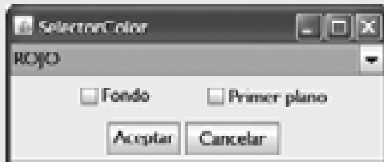
11.8 Cree la siguiente GUI. No tiene que proporcionar ningún tipo de funcionalidad.



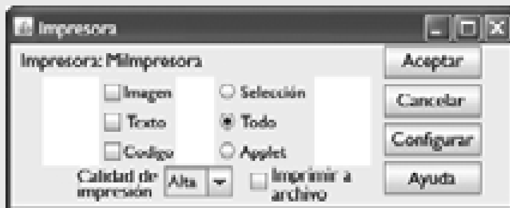
11.9 Cree la siguiente GUI. No tiene que proporcionar ningún tipo de funcionalidad.



11.10 Cree la siguiente GUI. No tiene que proporcionar ningún tipo de funcionalidad.



11.11 Cree la siguiente GUI. No tiene que proporcionar ningún tipo de funcionalidad.



11.12 Escriba una aplicación de conversión de temperatura, que convierta de grados Fahrenheit a Centígrados. La temperatura en grados Fahrenheit deberá introducirse desde el teclado (mediante un objeto `JTextField`). Debe usarse un objeto `JLabel` para mostrar la temperatura convertida. Use la siguiente fórmula para la conversión:

$$Celsius = \frac{5}{9} \times (Fahrenheit - 32)$$

11.13 Mejore la aplicación de conversión de temperatura del ejercicio 11.12, agregando la escala de temperatura Kelvin. Además, la aplicación debe permitir al usuario realizar conversiones entre dos escalas cualesquiera. Use la siguiente fórmula para la conversión entre Kelvin y Centígrados (además de la fórmula del ejercicio 11.12):

$$Kelvin = Centígrados + 273.15$$

11.14 Escriba una aplicación que muestre los eventos según vayan ocurriendo en un objeto `JTextArea`. Proporcione un objeto `JComboBox` con un mínimo de cuatro elementos. El usuario deberá ser capaz de seleccionar del objeto `JComboBox` un evento a vigilar. Cuando ocurra ese evento específico, muestre información acerca del mismo en el objeto `JTextArea`. Use el método `toString` en el objeto evento para convertirlo en una representación de cadena.

11.15 Escriba una aplicación que juegue a "adivinar el número" de la siguiente manera: su aplicación debe elegir el número a adivinar, seleccionando un entero al azar en el rango de 1 a 1000. La aplicación entonces deberá mostrar lo siguiente en una etiqueta:

```
Tengo un numero entre 1 y 1000. Puede usted adivinarlo?
Por favor escriba su primer intento.
```

Debe usarse un objeto `JTextField` para introducir el intento. A medida que se introduzca cada intento, el color de fondo deberá cambiar ya sea a rojo o azul. Rojo indica que el usuario se está "acercando" y azul indica que el usuario se está "alejando". Un objeto `JLabel` deberá mostrar el mensaje `"Demasiado alto"` o `"Demasiado bajo"` para ayudar al usuario a tratar de adivinar correctamente el número. Cuando el usuario adivine correctamente, deberá mostrarse el mensaje `"Correcto!"`, y el objeto `JTextField` utilizado para la entrada deberá cambiar para que no pueda editarse.

Debe proporcionarse un objeto JButton para permitir al usuario jugar de nuevo. Cuando se haga clic en el objeto JButton, deberá generarse un nuevo número aleatorio y el objeto JTextField de entrada deberá cambiar para poder editarse otra vez.

11.16 A menudo es conveniente mostrar los eventos que ocurren durante la ejecución de un programa. Esto puede ayudarle a comprender cuándo ocurren los eventos y cómo se generan. Escriba una aplicación que permita al usuario generar y procesar cada uno de los eventos descritos en este capítulo. La aplicación deberá proporcionar métodos de las interfaces ActionListener, ItemListener, ListSelectionListener, MouseListener, MouseMotionListener y KeyListener, para mostrar mensajes cuando ocurran los eventos. Use el método toString para convertir los objetos evento que se reciban en cada manejador de eventos, en un objeto String que pueda mostrarse en pantalla. El método toString crea un objeto String que contiene toda la información del objeto evento.

11.17 Modifique la aplicación de la sección 6.10 para proporcionar una GUI que permita al usuario hacer clic en un objeto JButton para tirar los dados. La aplicación debe también mostrar cuatro objetos JLabel y cuatro objetos JTextField, con un objeto JLabel para cada objeto JTextField. Los objetos JTextField deben usarse para mostrar los valores de cada dado, y la suma de los dados después de cada tiro. El punto debe mostrarse en el cuarto objeto JTextField cuando el usuario no gane o pierda en el primer tiro, y debe seguir mostrándose hasta que el usuario pierda el juego.

(Opcional) Ejercicio del ejemplo práctico de GUI y gráficos: expansión de la interfaz

18.18 En este ejercicio, implementará una aplicación de GUI que utiliza la jerarquía MiFigura del ejercicio 10.2 del ejemplo práctico de GUI, para crear una aplicación de dibujo interactiva. Debe crear dos clases para la GUI y proporcionar una clase de prueba para iniciar la aplicación. Las clases de la jerarquía MiFigura no requieren modificaciones adicionales.

La primera clase a crear es una subclase de JPanel llamada PanelDibujo, la cual representa el área en la cual el usuario dibuja las figuras. La clase PanelDibujo debe tener las siguientes variables de instancia:

a) Un arreglo llamado figuras de tipo MiFigura, que almacene todas las figuras que dibuje el usuario.
b) Una variable entera llamada cuentaFiguras, que cuente el número de figuras en el arreglo.
c) Una variable entera llamada tipoFigura, que determine el tipo de la figura a dibujar.
d) Un objeto MiFigura llamado figuraActual, que represente la figura actual que está dibujando el usuario.
e) Un objeto Color llamado colorActual, que represente el color del dibujo actual.
f) Una variable bolean llamada figuraRellena, que determine si se va a dibujar una figura rellena.
g) Un objeto JLabel llamado etiquetaEstado, que represente a la barra de estado. Esta barra deberá mostrar las coordenadas de la posición actual del ratón.

La clase PanelDibujo también debe declarar los siguientes métodos:

a) El método sobrescrito paintComponent, que dibuja las figuras en el arreglo. Use la variable de instancia cuentaFiguras para determinar cuántas figuras hay que dibujar. El método paintComponent también debe llamar al método draw de figuraActual, siempre y cuando figuraActual no sea null.
b) Métodos *establecer* para tipoFigura, colorActual y figuraRellena.
c) El método borrarUltimaFigura debe borrar la última figura dibujada, decrementando la variable de instancia cuentaFiguras. Asegúrese de que cuentaFiguras nunca sea menor que cero.
d) El método borrarDibujo debe eliminar todas las figuras en el dibujo actual, estableciendo cuentaFiguras en cero.

Los métodos borrarUltimaFigura y borrarDibujo deben llamar al método repaint (heredado de Jpanel) para actualizar el dibujo en el objeto PanelDibujo, indicando que el sistema nunca debe llamar al método paintComponent.

La clase PanelDibujo también debe proporcionar el manejo de eventos, para permitir al usuario dibujar con el ratón. Cree una clase interna individual que extienda a MouseAdapter e implemente MouseMotionListener para manejar todos los eventos de ratón en una clase.

En la clase interna, sobrescriba el método mousePressed de manera que asigne a figuraActual una nueva figura del tipo especificado por tipoFigura, y que inicialice ambos puntos con la posición del ratón. A continuación, sobrescriba el método mouseReleased para terminar de dibujar la figura actual y colocarla en el arreglo. Establezca el segundo punto de figuraActual con la posición actual del ratón y agregue figuraActual al arreglo. La variable de instancia cuentaFiguras determina el índice de inserción. Establezca figuraActual a null y llame al método repaint para actualizar el dibujo con la nueva figura.

Sobrescriba el método `mouseMoved` para establecer el texto de `etiquetaEstado`, de manera que muestre las coordenadas del ratón; esto actualizará la etiqueta con las coordenadas cada vez que el usuario mueva (pero no arrastre) el ratón dentro del objeto `PanelDibujo`. A continuación, sobrescriba el método `mouseDragged` de manera que establezca el segundo punto de `figuraActual` con la posición actual del ratón y llame al método `repaint`. Esto permitirá al usuario ver la figura mientras arrastra el ratón. Además, actualice el objeto `JLabel` en `mouseDragged` con la posición actual del ratón.

Cree un constructor para `PanelDibujo` que tenga un solo parámetro `JLabel`. En el constructor, inicialice `etiquetaEstado` con el valor que se pasa al parámetro. Además, inicialice el arreglo `figuras` con 100 entradas, `cuentaFiguras` con `0`, `tipoFigura` con el valor que represente a una línea, `figuraActual` con `null` y `colorActual` con `Color.BLACK`. El constructor deberá entonces establecer el color de fondo del objeto `PanelDibujo` a `Color.WHITE`. y registrar a `MouseListener` y `MouseMotionListener`, de manera que el objeto `JPanel` maneje los eventos de ratón en forma apropiada.

A continuación, cree una subclase de `JFrame` llamada `MarcoDibujo`, que proporcione una GUI que permita al usuario controlar varios aspectos del dibujo. Para el esquema del objeto `MarcoDibujo`, recomendamos `BorderLayout`, con los componentes en la región `NORTH`, el panel de dibujo principal en la región `CENTER` y una barra de estado en la región `SOUTH`, como en la figura 11.49. En el panel superior, cree los componentes que se listan a continuación. El manejador de eventos de cada componente deberá llamar al método apropiado en la clase `PanelDibujo`.

a) Un botón para deshacer la última figura que se haya dibujado.
b) Un botón para borrar todas las figuras del dibujo.
c) Un cuadro combinado para seleccionar el color de los 13 colores predefinidos.
d) Un cuadro combinado para seleccionar la figura a dibujar.
e) Una casilla de verificación que especifique si una figura debe estar rellena o sin relleno.

Declare y cree los componentes de la interfaz en el constructor de `MarcoDibujo`. Necesitará crear la barra de estado `JLabel` antes de crear el objeto `PanelDibujo`, de manera que pueda pasar el objeto `JLabel` como argumento para el constructor de `PanelDibujo`. Por último, cree una clase de prueba para inicializar y mostrar el objeto `Marco-Dibujo` para ejecutar la aplicación.

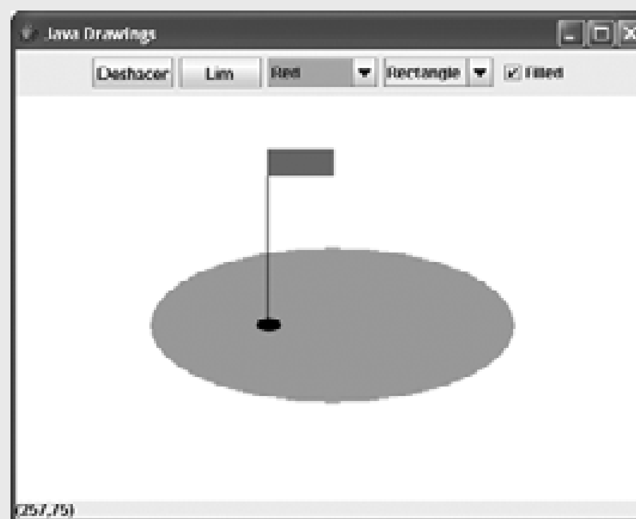


Figura 11.49 | Interfaz para dibujar figuras.

12

Gráficos y Java 2D™



Una imagen vale más que mil palabras.

—Proverbio chino

Hay que tratar a la naturaleza en términos del cilindro, de la esfera, del cono, todo en perspectiva.

—Paul Cézanne

Los colores, al igual que las características, siguen los cambios de las emociones.

—Pablo Picasso

Nada se vuelve real sino hasta que se experimenta; incluso un proverbio no será proverbio para usted, sino hasta que su vida lo haya ilustrado.

—John Keats

OBJETIVOS

En este capítulo aprenderá a:

- Comprender los contextos y los objetos de gráficos.
- Entender y manipular los colores.
- Comprender y manipular las fuentes.
- Usar métodos de la clase `Graphics` para dibujar líneas, rectángulos, rectángulos con esquinas redondeadas, rectángulos tridimensionales, óvalos, arcos y polígonos.
- Utilizar métodos de la clase `Graphics2D` de la API `Java 2D` para dibujar líneas, rectángulos, rectángulos con esquinas redondeadas, elipses, arcos y rutas en general.
- Especificar las características `Paint` y `Stroke` de las figuras mostradas con `Graphics2D`.

Plan general

12.1 Introducción

En este capítulo veremos varias de las herramientas de Java para dibujar figuras bidimensionales, controlar colores y fuentes. Uno de los principales atractivos de Java era su soporte para gráficos, el cual permitía a los programadores mejorar la apariencia visual de sus aplicaciones. Ahora, Java contiene muchas más herramientas sofisticadas de dibujo como parte de la API Java 2D™. Comenzaremos este capítulo con una introducción a muchas de las herramientas de dibujo originales de Java. Después presentaremos varias de las más poderosas herramientas de Java 2D, como el control del estilo de líneas utilizadas para dibujar figuras y el control del relleno de las figuras con colores y patrones. [*Nota:* ya hemos cubierto varios de los conceptos de este capítulo en el ejemplo práctico opcional de GUI y gráficos de los capítulos 3 a 10. Por lo tanto, parte del material será repetitivo si usted leyó el ejemplo práctico; sin embargo, si no lo ha leído, nos es necesario para comprender este capítulo].

En la figura 12.1 se muestra una porción de la jerarquía de clases de Java que incluye varias de las clases de gráficos básicas y las clases e interfaces de la API Java2 que cubriremos en este capítulo. La clase **`Color`** contiene métodos y constantes para manipular los colores. La clase `JComponent` contiene el método `paintComponent`, que se utiliza para dibujar gráficos en un componente. La clase **`Font`** contiene métodos y constantes para manejar los tipos de letras. La clase **`FontMetrics`** contiene métodos para obtener información sobre los tipos de letras. La clase `Graphics` contiene métodos para dibujar cadenas, líneas, rectángulos y demás figuras. La clase **`Graphics2D`**, que extiende a la clase `Graphics`, se utiliza para dibujar con la API Java 2D. La clase **`Polygon`** contiene métodos para crear polígonos. La mitad inferior de la figura muestra varias clases e interfaces de la API Java 2D. La clase **`BasicStroke`** ayuda a especificar las características de dibujo de las líneas. Las clases **`GradientPaint`** y **`TexturePaint`** ayudan a especificar las características para rellenar figuras con colores o patrones. Las clases `GeneralPath`, `Line2D`, `Arc2D`, `Ellipse2D`, `Rectangle2D` y `RoundRectangle2D` representan varias figuras de Java 2D. [*Nota:* empezaremos el capítulo hablando sobre las herramientas de gráficos originales de Java, y después pasaremos a la API Java 2D. Ahora, las clases que formaron parte de las herramientas de gráficos originales de Java se consideran parte de la API Java 2D].

Para empezar a dibujar en Java, primero debemos entender su **sistema de coordenadas** (figura 12.2), el cual es un esquema para identificar a cada uno de los posibles puntos en la pantalla. De manera predeterminada, la esquina superior izquierda de un componente de la GUI (como una ventana) tiene las coordenadas (0,0). Un par de coordenadas está compuesto por una **coordenada *x*** (la **coordenada horizontal**) y una **coordenada *y*** (la **coordenada vertical**). La coordenada *x* es la distancia horizontal que se desplaza hacia la derecha, desde la parte izquierda de la pantalla. La coordenada *y* es la distancia vertical que se desplaza hacia abajo, desde la parte superior de la pantalla. El **eje *x*** describe cada una de las coordenadas horizontales, y el **eje *y*** describe cada una de las coordenadas verticales. Las coordenadas se utilizan para indicar en dónde deben mostrarse los gráficos en una pantalla. Las unidades de las coordenadas se miden en **píxeles** ("elementos de imagen"). Un píxel es la unidad más pequeña de resolución de un monitor de computadora.



Tip de portabilidad 12.1

Existen distintos tipos de monitores de computadora con distintas resoluciones (es decir, la densidad de los píxeles varía). Esto puede hacer que los gráficos aparezcan de distintos tamaños en distintos monitores, o en el mismo monitor con distintas configuraciones.

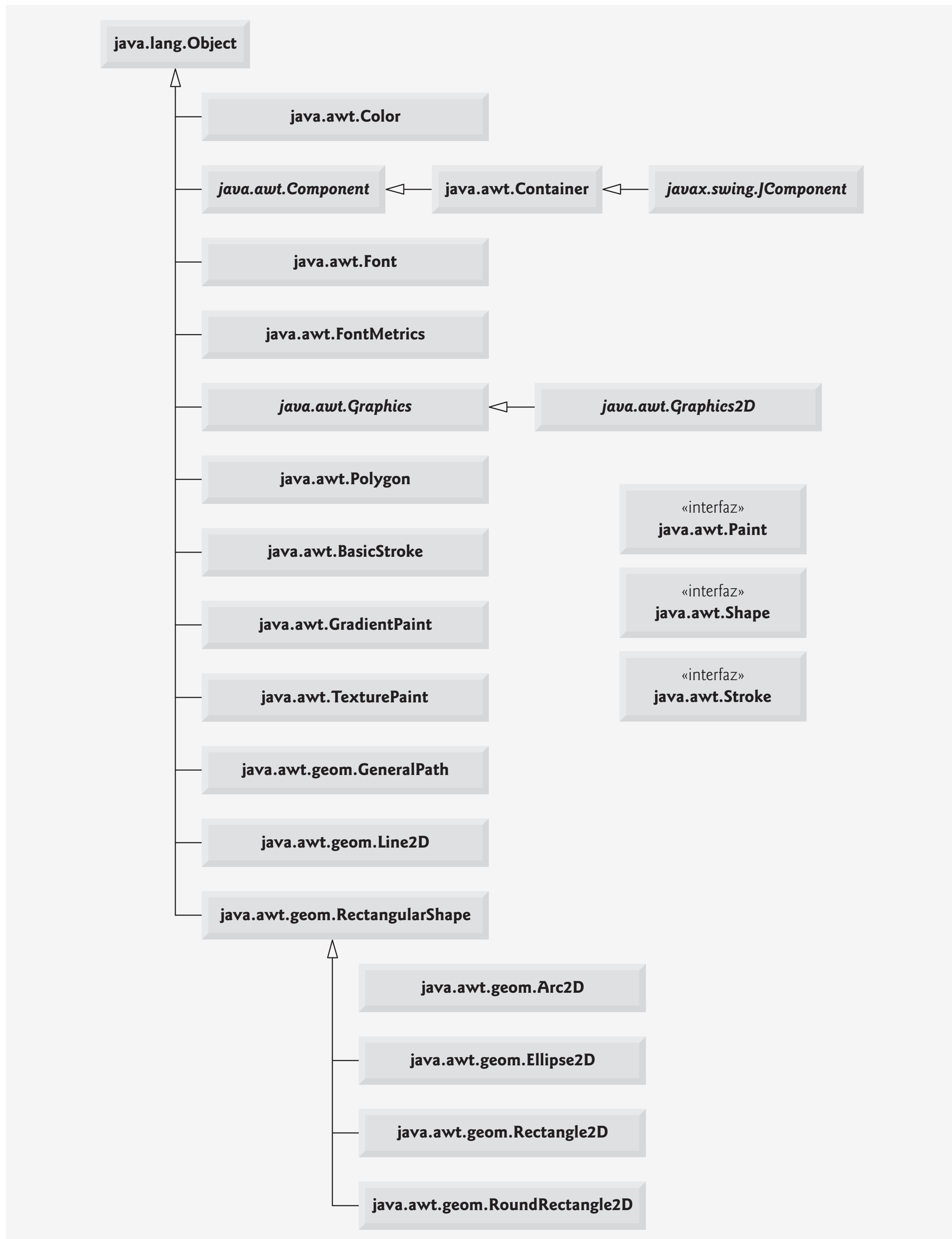


Figura 12.1 | Clases e interfaces utilizadas en este capítulo, provenientes de las herramientas de gráficos originales de Java y de la API Java2D. [*Nota:* la clase `Object` aparece aquí debido a que es la superclase de la jerarquía de clases de Java. Además, las clases `abstract` aparecen en cursiva].

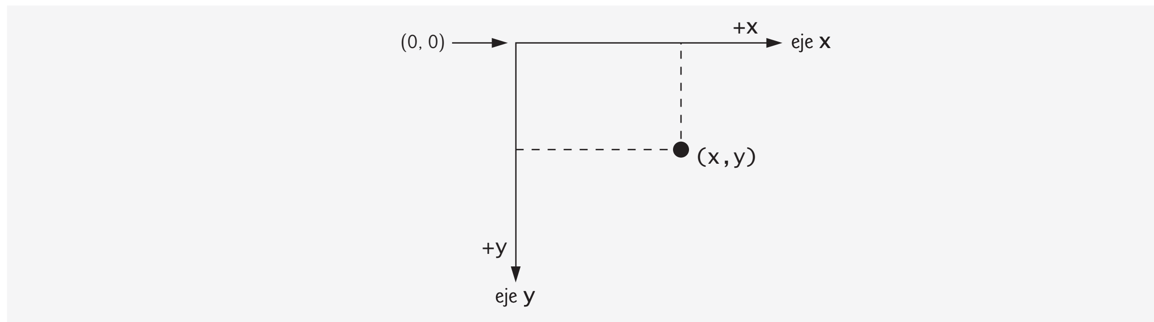


Figura 12.2 | Sistema de coordenadas de Java. Las unidades se miden en píxeles.

12.2 Contextos y objetos de gráficos

Un **contexto de gráficos** permite dibujar en la pantalla. Un objeto `Graphics` administra un contexto de gráficos y dibuja píxeles en la pantalla que representan texto y otros objetos gráficos (como líneas, elipses, rectángulos y otros polígonos). Los objetos `Graphics` contienen métodos para dibujar, manipular tipos de letra, manipular colores y varias cosas más.

La clase `Graphics` es una clase `abstract` (es decir, no pueden instanciarse objetos `Graphics`). Esto contribuye a la portabilidad de Java. Como el dibujo se lleva a cabo de manera distinta en cada plataforma que soporta a Java, no puede haber sólo una implementación de las herramientas de dibujo en todos los sistemas. Por ejemplo, las herramientas de gráficos que permiten a una PC con Microsoft Windows dibujar un rectángulo, son distintas de las herramientas de gráficos que permiten a una estación de trabajo Linux dibujar un rectángulo; y ambas son distintas de las herramientas de gráficos que permiten a una Macintosh dibujar un rectángulo. Cuando Java se implementa en cada plataforma, se crea una subclase de `Graphics` que implementa las herramientas de dibujo. Esta implementación está oculta para nosotros por medio de la clase `Graphics`, la cual proporciona la interfaz que nos permite utilizar gráficos de una manera independiente de la plataforma.

La clase `Component` es la superclase para muchas de las clases en el paquete `java.awt`. (En el capítulo 11 presentamos la clase `Component`). La clase `JComponent`, que hereda directamente de `Component`, contiene un método llamado `paintComponent`, que puede utilizarse para dibujar gráficos. El método `paintComponent` toma un objeto `Graphics` como argumento. El sistema pasa este objeto al método `paintComponent` cuando se requiere volver a pintar un componente ligero de Swing. El encabezado del método `paintComponent` es:

```
public void paintComponent( Graphics g )
```

El parámetro `g` recibe una referencia a una instancia de la subclase específica del sistema que `Graphics` extiende. Tal vez a usted le parezca conocido el encabezado del método anterior; es el mismo que utilizamos en algunas de las aplicaciones del capítulo 11. En realidad, la clase `JComponent` es una superclase de `JPanel`. Muchas herramientas de la clase `JPanel` son heredadas de la clase `JComponent`.

El método `paintComponent` raras veces es llamado directamente por el programador, ya que el dibujo de gráficos es un proceso controlado por eventos. Cuando se ejecuta una aplicación de GUI, el contenedor de la aplicación llama al método `paintComponent` para cada componente ligero, a medida que se muestra la GUI en pantalla. Para que `paintComponent` sea llamado de nuevo, debe ocurrir un evento (como cubrir y descubrir el componente con otra ventana).

Si el programador necesita hacer que se ejecute `paintComponent` (es decir, si desea actualizar los gráficos dibujados en el componente de Swing), se hace una llamada al método **`repaint`**, que todos los objetos `JComponent` heredan indirectamente de la clase `Component` (paquete `java.awt`). El método `repaint` se llama con frecuencia para solicitar una llamada al método `paintComponent`. El encabezado para `repaint` es:

```
public void repaint()
```

12.3 Control de colores

La clase `Color` declara los métodos y las constantes para manipular los colores en un programa de Java. Las constantes de colores previamente declaradas se sintetizan en la figura 12.3, y varios métodos y constructores para los

colores se sintetizan en la figura 12.4. Observe que dos de los métodos de la figura 12.4 son métodos de `Graphics` que son específicos para los colores.

Constante de Color	Valor RGB
`public final static Color RED`	255, 0, 0
`public final static Color GREEN`	0, 255, 0
`public final static Color BLUE`	0, 0, 255
`public final static Color ORANGE`	255, 200, 0
`public final static Color PINK`	255, 175, 175
`public final static Color CYAN`	0, 255, 255
`public final static Color MAGENTA`	255, 0, 255
`public final static Color YELLOW`	255, 255, 0
`public final static Color BLACK`	0, 0, 0
`public final static Color WHITE`	255, 255, 255
`public final static Color GRAY`	128, 128, 128
`public final static Color LIGHT_GRAY`	192, 192, 192
`public final static Color DARK_GRAY`	64, 64, 64

Figura 12.3 | Constantes de `Color` y sus valores RGB.

Método	Descripción
Constructores y métodos de `Color`	
`public Color( int r, int g, int b )`	
	Crea un color basado en los componentes rojo, verde y azul, expresados como enteros de 0 a 255.
`public Color( float r, float g, float b )`	
	Crea un color basado en los componentes rojo, verde y azul, expresados como valores de punto flotante de 0.0 a 1.0.
`public int getRed()`	
	Devuelve un valor entre 0 y 255, el cual representa el contenido rojo.
`public int getGreen()`	
	Devuelve un valor entre 0 y 255, el cual representa el contenido verde.
`public int getBlue()`	
	Devuelve un valor entre 0 y 255, el cual representa el contenido azul.
Métodos de `Graphics` *para manipular objetos* `Color`	
`public Color getColor()`	
	Devuelve un objeto `Color` que representa el color actual para el contexto de gráficos.
`public void setColor( Color c )`	
	Establece el color actual para dibujar con el contexto de gráficos.

Figura 12.4 | Los métodos de `Color` y los métodos de `Graphics` relacionados con los colores.

Todo color se crea a partir de un componente rojo, uno verde y otro azul. En conjunto, a estos componentes se les llama **valores RGB**. Los tres componentes RGB pueden ser enteros en el rango de 0 a 255, o pueden ser valores de punto flotante en el rango de 0.0 a 1.0. El primer componente RGB especifica la cantidad de rojo, el segundo, de verde y el tercero, de azul. Entre mayor sea el valor RGB, mayor será la cantidad de ese color en particular. Java permite al programador seleccionar de entre 256 x 256 x 256 (o aproximadamente 16.7 millones de) colores. No todas las computadoras son capaces de mostrar todos estos colores. La computadora mostrará el color más cercano que pueda.

En la figura 12.4 se muestran dos de los constructores de la clase `Color` (uno que toma tres argumentos `int` y otro que toma tres argumentos `float`, en donde cada argumento especifica la cantidad de rojo, verde y azul). Los valores `int` deben estar en el rango de 0 a 255 y los valores `float` deben estar en el rango de 0.0 a 1.0. El nuevo objeto `Color` tendrá las cantidades de rojo, azul y verde que se especifiquen. Los métodos **`getRed`**, **`getGreen`** y **`getBlue`** de `Color` devuelven valores enteros de 0 a 255, los cuales representan la cantidad de rojo, verde y azul, respectivamente. El método **`getColor`** de `Graphics` devuelve un objeto `Color` que representa el color actual de dibujo. El método **`setColor`** de `Graphics` establece el color actual de dibujo.

Las figuras 12.5 y 12.6 demuestran varios métodos de la figura 12.4, al dibujar rectángulos rellenos y cadenas en varios colores distintos. Cuando la aplicación empieza a ejecutarse, se hace una llamada al método `paintComponent` de la clase `JPanelColor` (líneas 10 a 37 de la figura 12.5) para pintar la ventana. En la línea 17 se utiliza el método `setColor` de `Graphics` para establecer el color actual de dibujo. El método `setColor` recibe un objeto `Color`. La expresión `new Color( 255, 0, 0 )` crea un nuevo objeto `Color` que representa rojo (valor `255` para rojo y `0` para los valores azul y verde). En la línea 18 se utiliza el método **`fillRect`** de `Graphics` para dibujar un rectángulo relleno con el color actual. El método `fillRect` dibuja un rectángulo con base en sus cuatro argumentos. Los primeros dos valores enteros representan la coordenada *x* superior izquierda y la coordenada *y* superior izquierda, en donde el objeto `Graphics` empieza a dibujar el rectángulo. Los argumentos tercero y cuarto son enteros positivos que representan la anchura y la altura del rectángulo en píxeles, respectivamente. Un rectángulo que se dibuja usando el método `fillRect` se rellena con el color actual del objeto `Graphics`.

En la línea 19 se utiliza el método **`drawString`** de `Graphics` para dibujar un objeto `String` en el color actual. La expresión `g.getColor()` recupera el color actual del objeto `Graphics`. El objeto `Color` devuelto se concatena con la cadena `"RGB actual:"`, lo que produce una llamada implícita al método `toString` de la clase `Color`. La representación `String` de un objeto `Color` contiene el nombre de la clase y el paquete (`java.awt.Color`), además de los valores rojo, verde y azul.

```
 1  // Fig. 12.5: JPanelColor.java
 2  // Demostración de objetos Color.
 3  import java.awt.Graphics;
 4  import java.awt.Color;
 5  import javax.swing.JPanel;
 6
 7  public class JPanelColor extends JPanel
 8  {
 9     // dibuja rectángulos y objetos String en distintos colores
10     public void paintComponent( Graphics g )
11     {
12        super.paintComponent( g ); // llama al método paintComponent de la superclase
13
14        this.setBackground( Color.WHITE );
15
16        // establece nuevo color de dibujo, usando valores enteros
17        g.setColor( new Color( 255, 0, 0 ) );
18        g.fillRect( 15, 25, 100, 20 );
19        g.drawString( "RGB actual: " + g.getColor(), 130, 40 );
20
21        // establece nuevo color de dibujo, usando valores de punto flotante
22        g.setColor( new Color( 0.50f, 0.75f, 0.0f ) );
```

Figura 12.5 | Programa para imprimir texto. (Parte 1 de 2).

```java
      g.fillRect( 15, 50, 100, 20 );
      g.drawString( "RGB actual: " + g.getColor(), 130, 65 );

      // establece nuevo color de dibujo, usando objetos Color static
      g.setColor( Color.BLUE );
      g.fillRect( 15, 75, 100, 20 );
      g.drawString( "RGB actual: " + g.getColor(), 130, 90 );

      // muestra los valores RGB individuales
      Color color = Color.MAGENTA;
      g.setColor( color );
      g.fillRect( 15, 100, 100, 20 );
      g.drawString( "Valores RGB: " + color.getRed() + ", " +
         color.getGreen() + ", " + color.getBlue(), 130, 115 );
   } // fin del método paintComponent
} // fin de la clase JPanelColor
```

Figura 12.5 | Cambio de colores para dibujar. (Parte 2 de 2).

```java
// Fig. 12.6: MostrarColores.java
// Demostración de objetos Color.
import javax.swing.JFrame;

public class MostrarColores
{
   // ejecuta la aplicación
   public static void main( String args[] )
   {
      // crea marco para objeto JPanelColor
      JFrame frame = new JFrame( "Uso de colores" );
      frame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );

      JPanelColor jPanelColor = new JPanelColor(); // crea objeto JPanelColor
      frame.add( jPanelColor ); // agrega jPanelColor a marco
      frame.setSize( 400, 180 ); // establece el tamaño del marco
      frame.setVisible( true ); // muestra el marco
   } // fin de main
} // fin de la clase MostrarColores
```



Figura 12.6 | Creación de un objeto `JFrame` para mostrar colores en un objeto `JPanel`.



Observación de apariencia visual 12.1

Todos percibimos los colores de una forma distinta. Elija sus colores con cuidado, para asegurarse que su aplicación sea legible, tanto para las personas que pueden percibir el color, como para aquellas que no pueden ver ciertos colores. Trate de evitar usar muchos colores distintos, muy cerca unos de otros.

En las líneas 22 a 24 y 27 a 29 se llevan a cabo, nuevamente, las mismas tareas. En la línea 22 se utiliza el constructor de `Color` con tres argumentos `float` para crear el color verde oscuro (`0.50f` para rojo, `0.75f` para verde y `0.0f` para azul). Observe la sintaxis de los valores. La letra `f` anexada a una literal de punto flotante indica que la literal debe tratarse como de tipo `float`. De manera predeterminada, las literales de punto flotante se tratan como de tipo `double`.

En la línea 27 se establece el color actual de dibujo a una de las constantes de `Color` previamente declaradas (`Color.BLUE`). Las constantes de `Color` son `static`, por lo que se crean cuando la clase `Color` se carga en memoria, en tiempo de ejecución.

La instrucción de las líneas 35 y 36 hace llamadas a los métodos `getRed`, `getGreen` y `getBlue` de `Color` en la constante `Color.MAGENTA` previamente declarada. El método `main` de la clase `MostrarColores` (líneas 8 a 18 de la figura 12.6) crea el objeto `JFrame` que contendrá un objeto `ColorJPanel`, en donde se mostrarán los colores.



Observación de ingeniería de software 12.1

Para cambiar el color, debe crear un nuevo objeto `Color` (o utilizar una de las constantes de `Color` previamente declaradas). Al igual que los objetos `String`, los objetos `Color` son inmutables (no pueden modificarse).

El paquete `javax.swing` proporciona el componente de la GUI **`JColorChooser`** para permitir a los usuarios de aplicaciones seleccionar colores. La aplicación de las figuras 12.7 y 12.8 demuestra un cuadro de diálogo `JColorChooser`. Al hacer clic en el botón **Cambiar color**, aparece un cuadro de diálogo `JColorChooser`. Al seleccionar un color y oprimir el botón **Aceptar**, el color de fondo de la ventana de la aplicación cambia.

```java
// Fig. 12.7: MostrarColores2JFrame.java
// Selección de colores con JColorChooser.
import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JColorChooser;
import javax.swing.JPanel;

public class MostrarColores2JFrame extends JFrame
{
   private JButton cambiarColorJButton;
   private Color color = Color.LIGHT_GRAY;
   private JPanel coloresJPanel;

   // establece la GUI
   public MostrarColores2JFrame()
   {
      super( "Uso de JColorChooser" );

      // crea objeto JPanel para mostrar color
      coloresJPanel = new JPanel();
      coloresJPanel.setBackground( color );

      // establece cambiarColorJButton y registra su manejador de eventos
      cambiarColorJButton = new JButton( "Cambiar color" );
      cambiarColorJButton.addActionListener(

         new ActionListener() // clase interna anónima
         {
            // muestra JColorChooser cuando el usuario hace clic con el botón
```

Figura 12.7 | Cuadro de diálogo `JColorChooser`. (Parte 1 de 2).

```
34              public void actionPerformed( ActionEvent evento )
35              {
36                 color = JColorChooser.showDialog(
37                    MostrarColores2JFrame.this, "Seleccione un color", color );
38
39                 // establece el color predeterminado, si no se devuelve un color
40                 if ( color == null )
41                    color = Color.LIGHT_GRAY;
42
43                 // cambia el color de fondo del panel de contenido
44                 coloresJPanel.setBackground( color );
45              } // fin del método actionPerformed
46           } // fin de la clase interna anónima
47        ); // fin de la llamada a addActionListener
48
49        add( coloresJPanel, BorderLayout.CENTER ); // agrega coloresJPanel
50        add( cambiarColorJButton, BorderLayout.SOUTH ); // agrega botón
51
52        setSize( 400, 130 ); // establece el tamaño del marco
53        setVisible( true ); // muestra el marco
54     } // fin del constructor de MostrarColores2JFrame
55  } // fin de la clase MostrarColores2JFrame
```

Figura 12.7 | Cuadro de diálogo `JColorChooser`. (Parte 2 de 2).

La clase `JColorChooser` proporciona el método estático **`showDialog`**, el cual crea un objeto `JColorChooser`, lo adjunta a un cuadro de diálogo y lo muestra en pantalla. Las líneas 36 y 37 de la figura 12.7 invocan a este método para mostrar el cuadro de diálogo del selector de colores. El método `showDialog` devuelve el objeto `Color` seleccionado, o `null` si el usuario oprime **Cancelar** o cierra el cuadro de diálogo sin oprimir **Aceptar**. Este método recibe tres argumentos: una referencia a su objeto `Component` padre, un objeto `String` a mostrar en la barra de título del cuadro de diálogo y el `Color` inicial seleccionado para el cuadro de diálogo. El componente padre es una referencia a la ventana desde la que se muestra el cuadro de diálogo (en este caso el objeto `JFrame`, con el nombre de referencia `marco`). Este cuadro de diálogo estará centrado en el componente padre. Si el padre es `null`, entonces el cuadro de diálogo se centra en la pantalla. Mientras el cuadro de diálogo para seleccionar colores se encuentre en la pantalla, el usuario no podrá interactuar con el componente padre. A este tipo de cuadro de diálogo se le conoce como **cuadro de diálogo modal** (el cual se describirá en el capítulo 22, Componentes de la GUI: parte 2).

Una vez que el usuario selecciona un color, en las líneas 40 y 41 se determina si `color` es `null`, y de ser así `color` se establece en el valor predeterminado `Color.LIGHT_GRAY`. En la línea 44 se utiliza el método `setBackground` para cambiar el color de fondo del objeto `JPanel`. El método `setBackground` es uno de los muchos métodos de la clase `Component` que pueden utilizarse en la mayoría de los componentes de la GUI. Observe que el usuario puede seguir utilizando el botón **Cambiar color** para cambiar el color de fondo de la aplicación. La figura 12.8 contiene el método `main`, que ejecuta el programa.

```
// Fig. 12.8: MostrarColores2.java
// Selección de colores con JColorChooser.
import javax.swing.JFrame;

public class MostrarColores2
{
   // ejecuta la aplicación
   public static void main( String args[] )
   {
```

Figura 12.8 | Selección de colores con `JColorChooser`. (Parte 1 de 2).

```
        MostrarColores2JFrame aplicacion = new MostrarColores2JFrame();
        aplicacion.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
    } // fin de main
} // fin de la clase MostrarColores2
```

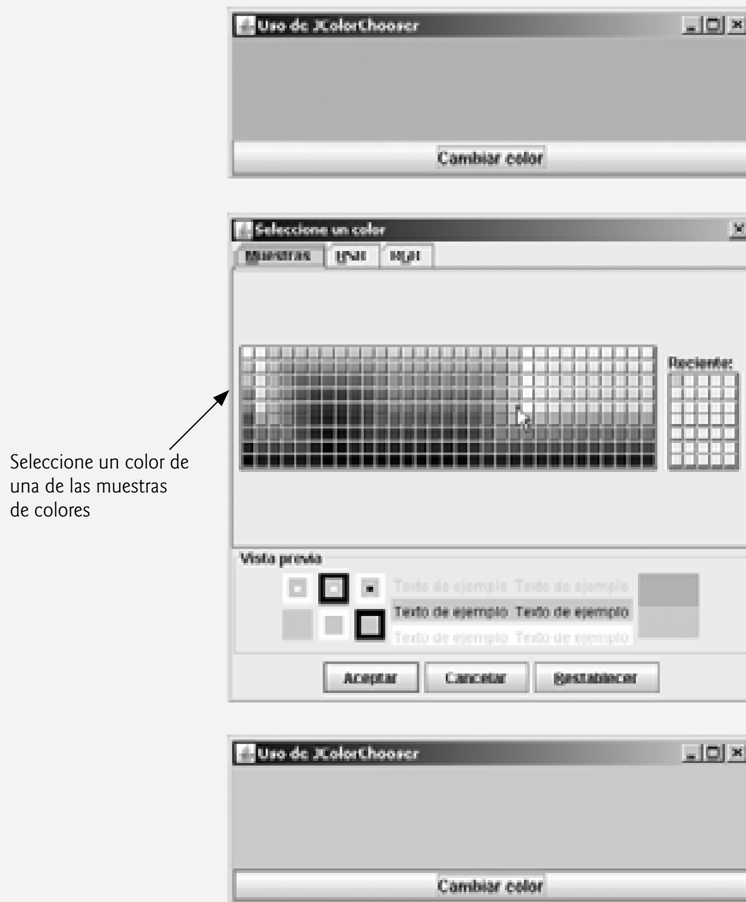


Figura 12.8 | Selección de colores con `JColorChooser`. (Parte 2 de 2).

La segunda captura de pantalla de la figura 12.8 muestra el cuadro de diálogo `JColorChooser` predeterminado, que permite al usuario seleccionar un color de una variedad de **muestras de colores**. Observe que en realidad hay tres fichas en la parte superior del cuadro de diálogo: **Muestras**, **HSB** y **RGB**. Estas fichas representan tres distintas formas de seleccionar un color. La ficha **HSB** le permite seleccionar un color con base en **matiz** (hue), **saturación** (saturation) y **brillo** (brightness): valores que se utilizan para definir la cantidad de luz en un color. No hablaremos sobre los valores HSB. Para obtener más información sobre matiz, saturación y brillo, visite `whatis.techtarget.com/definition/0,,sid9_gci212262,00.html`. La ficha **RGB** le permite seleccionar un color mediante el uso de controles deslizables para seleccionar los componentes rojo, verde y azul del color. Las fichas **HSB** y **RGB** se muestran en la figura 12.9.

12.4 Control de tipos de letra

En esta sección presentaremos los métodos y constantes para controlar los tipos de letras. La mayoría de los métodos y constantes de tipos de letra son parte de la clase `Font`. Algunos métodos de la clase `Font` y la clase `Graphics` se sintetizan en la figura 12.10.

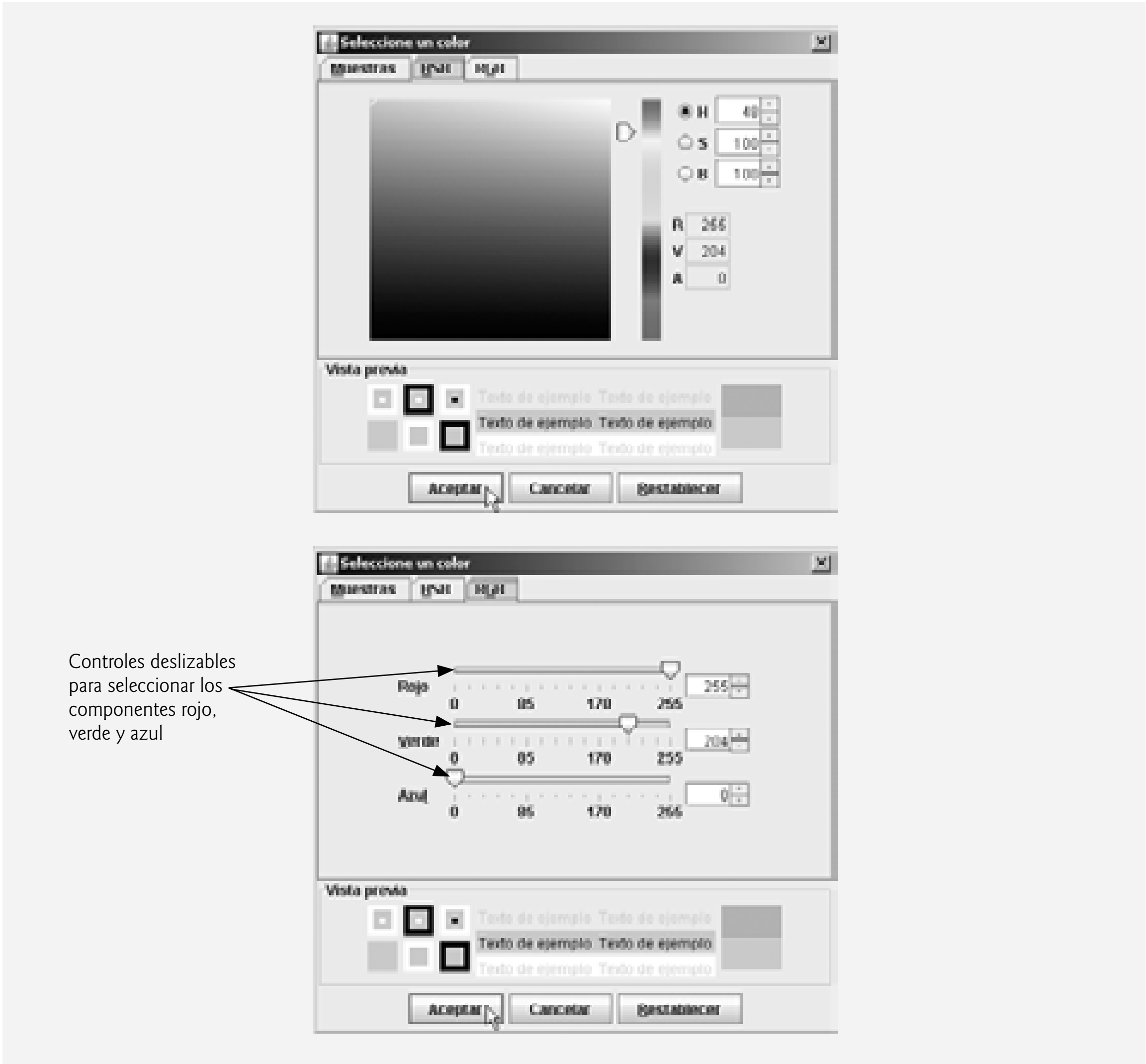


Figura 12.9 | Las fichas **HSB** y **RGB** del cuadro de diálogo `JColorChooser`.

Método o constante	Descripción
Constantes, constructores y métodos de `Font`	
`public final static int PLAIN`	Constante que representa un estilo de tipo de letra simple.
`public final static int BOLD`	Constante que representa un estilo de tipo de letra en negritas.
`public final static int ITALIC`	Constante que representa un estilo de tipo de letra en cursivas.
`public Font( String nombre, int estilo, int tamaño )`	Crea un objeto `Font` con el nombre de tipo de letra, estilo y tamaño especificados.
`public int getStyle()`	Devuelve un valor entero que indica el estilo actual de tipo de letra.
`public int getSize()`	Devuelve un valor entero que indica el tamaño actual del tipo de letra.

Figura 12.10 | Métodos y constantes relacionados con `Font`. (Parte 1 de 2).

Método o constante	Descripción
Constantes, constructores y métodos de `Font`	
`public String getName()`	Devuelve el nombre actual del tipo de letra, como una cadena.
`public String getFamily()`	Devuelve el nombre de la familia del tipo de letra, como una cadena.
`public boolean isPlain()`	Devuelve `true` si el tipo de letra es simple; `false` en caso contrario.
`public boolean isBold()`	Devuelve `true` si el tipo de letra está en negritas; `false` en caso contrario.
`public boolean isItalic()`	Devuelve `true` si el tipo de letra está en cursivas; `false` en caso contrario.
Métodos de Graphics para manipular objetos `Font`	
`public Font getFont()`	Devuelve la referencia a un objeto `Font` que representa el tipo de letra actual.
`public void setFont( Font f )`	Establece el tipo de letra actual al tipo de letra, estilo y tamaño especificados por la referencia `f` al objeto `Font`.

Figura 12.10 | Métodos y constantes relacionados con `Font`. (Parte 2 de 2).

El constructor de la clase `Font` recibe tres argumentos: el **nombre del tipo de letra**, su **estilo** y su **tamaño**. El nombre del tipo de letra es cualquier tipo de letra soportado por el sistema en el que se esté ejecutando el programa, como los tipos de letra estándar de Java `Monospaced`, `SansSerif` y `Serif`. El estilo de tipo de letra es **`Font.PLAIN`** (simple), **`Font.ITALIC`** (cursivas) o **`Font.BOLD`** (negritas); cada uno es un campo `static` de la clase `Font`. Los estilos de los tipos de letra pueden usarse combinados (por ejemplo, `Font.ITALIC + Font.BOLD`). El tamaño del tipo de letra se mide en puntos. Un **punto** es 1/72 de una pulgada. El método **`setFont`** de `Graphics` establece el tipo de letra a dibujar en ese momento (el tipo de letra en el cual se mostrará el texto) en base a su argumento `Font`.



Tip de portabilidad 12.2

El número de tipos de letra varía enormemente entre sistemas. Java proporciona cinco nombres de tipos de letras (`Serif`, `Monospaced`, `SansSerif`, `Dialog` y `DialogInput`) que pueden usarse en todas las plataformas de Java. El entorno en tiempo de ejecución de Java (JRE) en cada plataforma asigna estos nombres de tipos de letras lógicos a los tipos de letras que están realmente instalados en la plataforma. Los tipos de letras reales que se utilicen pueden variar de una plataforma a otra.

La aplicación de las figuras 12.11 y 12.12 muestra texto en cuatro tipos de letra distintos, con cada tipo de letra en diferente tamaño. La figura 12.11 utiliza el constructor de `Font` para inicializar objetos `Font` (en las líneas 16, 20, 24 y 29) que se pasan al método `setFont` de `Graphics` para cambiar el tipo de letra para dibujar. Cada llamada al constructor de `Font` pasa un nombre de tipo de letra (`Serif`, `Monospaced`, o `SansSerif`) como una cadena, un estilo de tipo de letra (`Font.PLAIN`, `Font.ITALIC` o `Font.BOLD`) y un tamaño de tipo de letra. Una vez que se invoca el método `setFont` de `Graphics`, todo el texto que se muestre después de la llamada aparecerá en el nuevo

```
1  // Fig. 12.11: FontJPanel.java
2  // Muestra cadenas en distintos tipos de letra y colores.
3  import java.awt.Font;
4  import java.awt.Color;
5  import java.awt.Graphics;
6  import javax.swing.JPanel;
7
8  public class FontJPanel extends JPanel
9  {
10    // muestra objetos String en distintos tipos de letra y colores
```

Figura 12.11 | El método `setFont` de `Graphics` cambia el tipo de letra para dibujar. (Parte 1 de 2).

```
    public void paintComponent( Graphics g )
    {
       super.paintComponent( g ); // llama al método paintComponent de la superclase

       // establece el tipo de letra a Serif (Times), negrita, 12 puntos y dibuja una cadena
       g.setFont( new Font( "Serif", Font.BOLD, 12 ) );
       g.drawString( "Serif 12 puntos, negrita.", 20, 50 );

       // establece el tipo de letra a Monospaced (Courier), cursiva, 24 puntos y dibuja una cadena
       g.setFont( new Font( "Monospaced", Font.ITALIC, 24 ) );
       g.drawString( "Monospaced 24 puntos, cursiva.", 20, 70 );

       // establece el tipo de letra a SansSerif (Helvetica), simple, 14 puntos y dibuja una cadena
       g.setFont( new Font( "SansSerif", Font.PLAIN, 14 ) );
       g.drawString( "SansSerif 14 puntos, simple.", 20, 90 );

       // establece el tipo de letra a Serif (Times), negrita/cursiva, 18 puntos y dibuja una cadena
       g.setColor( Color.RED );
       g.setFont( new Font( "Serif", Font.BOLD + Font.ITALIC, 18 ) );
       g.drawString( g.getFont().getName() + " " + g.getFont().getSize() +
          " puntos, negrita cursiva.", 20, 110 );
    } // fin del método paintComponent
 } // fin de la clase FontJPanel
```

Figura 12.11 | El método `setFont` de `Graphics` cambia el tipo de letra para dibujar. (Parte 2 de 2).

```
// Fig. 12.12: TiposDeLetra.java
// Uso de tipos de letra.
import javax.swing.JFrame;

public class TiposDeLetra
{
   // ejecuta la aplicación
   public static void main( String args[] )
   {
      // crea marco para FontJPanel
      JFrame marco = new JFrame( "Uso de tipos de letra" );
      marco.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );

      FontJPanel fontJPanel = new FontJPanel(); // crea objeto FontJPanel
      marco.add( fontJPanel ); // agrega objeto fontJPanel al marco
      marco.setSize( 475, 170 ); // establece el tamaño del marco
      marco.setVisible( true ); // muestra el marco
   } // fin de main
} // fin de la clase TiposDeLetra
```



Figura 12.12 | Creación de un objeto `JFrame` para mostrar tipos de letra.

tipo de letra hasta que éste se modifique. La información de cada tipo de letra se muestra en las líneas 17, 21, 25, 30 y 31, usando el método drawString. Observe que la coordenada que se pasa a drawString corresponde a la esquina inferior izquierda de la línea base del tipo de letra. En la línea 28 se cambia el color de dibujo a rojo, por lo que la siguiente cadena que se muestra aparece en color rojo. En las líneas 30 a 31 se muestra información acerca del objeto Font final. El método **getFont** de la clase Graphics devuelve un objeto Font que representa el tipo de letra actual. El método **getName** devuelve el nombre del tipo de letra actual como una cadena. El método **getSize** devuelve el tamaño del tipo de letra, en puntos.

La figura 12.12 contiene el método main, que crea un objeto JFrame. Agregamos un objeto FontJPanel a este objeto JFrame (línea 15), el cual muestra los gráficos creados en la figura 12.11.



Observación de ingeniería de software 12.2

Para cambiar el tipo de letra, debe crear un nuevo objeto Font. *Los objetos* Font *son inmutables; la clase* Font *no tiene métodos establecer para modificar las características del tipo de letra actual.*

Métrica de los tipos de letra

En ocasiones es necesario obtener información acerca del tipo de letra actual para dibujar, como el nombre, el estilo y el tamaño del tipo de letra. En la figura 12.10 se sintetizan varios métodos de Font que se utilizan para obtener información sobre el tipo de letra. El método **getStyle** devuelve un valor entero que representa el estilo actual. El valor entero devuelto puede ser Font.PLAIN, Font.ITALIC, Font.BOLD o la combinación de Font.ITALIC y Font.BOLD. El método **getFamily** devuelve el nombre de la familia a la que pertenece el tipo de letra actual. El nombre de la familia del tipo de letra es específico de la plataforma. También hay métodos de Font disponibles para probar el estilo del tipo de letra actual, los cuales se sintetizan también en la figura 12.10. Los métodos **isPlain**, **isBold** e **isItalic** devuelven true si el estilo del tipo de letra actual es simple, negrita o cursiva, respectivamente.

Algunas veces es necesario conocer información precisa acerca de la métrica de un tipo de letra, como la **altura**, el **descendente** (la distancia entre la base de la línea y el punto inferior del tipo de letra), el **ascendente** (la cantidad que se eleva un carácter por encima de la base de la línea) y el **interlineado** (la diferencia entre el descendente de una línea de texto y el ascendente de la línea de texto que está debajo; es decir, el espaciamiento entre líneas). En la figura 12.13 se muestran algunos elementos comunes de la **métrica de los tipos de letras**.

La clase **FontMetrics** declara varios métodos para obtener información métrica de los tipos de letra. En la figura 12.14 se sintetizan estos métodos, junto con el método **getFontMetrics** de la clase Graphics. La aplicación de las figuras 12.15 y 12.16 utiliza los métodos de la figura 12.14 para obtener la información métrica de dos tipos de letra.

En la línea 15 de la figura 12.15 se crea y se establece el tipo de letra actual para dibujar en SansSerif, negrita, 12 puntos. En la línea 16 se utiliza el método getFontMetrics de Graphics para obtener el objeto FontMetrics del tipo de letra actual. En la línea 17 se imprime la representación String del objeto Font devuelto por g.getFont(). En las líneas 18 a 21 se utilizan los métodos de FontMetrics para obtener el ascendente, descendente, altura e interlineado del tipo de letra.

En la línea 23 se crea un nuevo tipo de letra Serif, cursiva, 14 puntos. En la línea 24 se utiliza una segunda versión del método getFontMetrics de Graphics, la cual recibe un argumento Font y devuelve su correspondiente objeto FontMetrics. En las líneas 27 a 30 se obtiene el ascendente, descendente, altura e interlineado de ese tipo de letra. Observe que la métrica es ligeramente distinta para cada uno de los tipos de letra.

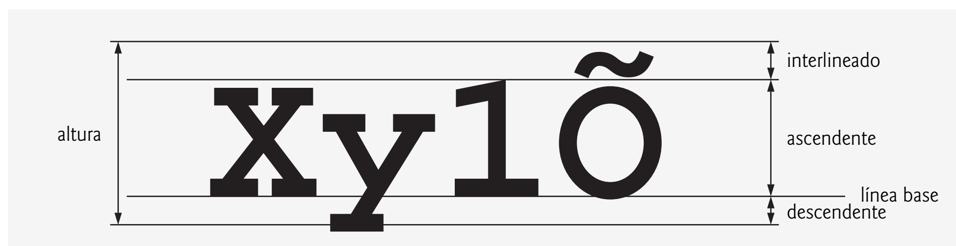


Figura 12.13 | Métrica de los tipos de letra.

Método	Descripción
Métodos de `FontMetrics`	
`public int getAscent()`	
	Devuelve un valor que representa el ascendente de un tipo de letra, en puntos.
`public int getDescent()`	
	Devuelve un valor que representa el descendente de un tipo de letra, en puntos.
`public int getLeading()`	
	Devuelve un valor que representa el interlineado de un tipo de letra, en puntos.
`public int getHeight()`	
	Devuelve un valor que representa la altura de un tipo de letra, en puntos.
Métodos de `Graphics` *para obtener la métrica de un tipo de letra*	
`public FontMetrics getFontMetrics()`	
	Devuelve el objeto `FontMetrics` para el objeto `Font` actual para dibujar.
`public FontMetrics getFontMetrics( Font f )`	
	Devuelve el objeto `FontMetrics` para el argumento `Font` especificado.

Figura 12.14 | Métodos de `FontMetrics` y `Graphics` para obtener la métrica de los tipos de letra.

```
// Fig. 12.15: MetricaJPanel.java
// Métodos de FontMetrics y Graphics útiles para obtener la métrica de los tipos de
letra.
import java.awt.Font;
import java.awt.FontMetrics;
import java.awt.Graphics;
import javax.swing.JPanel;

public class MetricaJPanel extends JPanel
{
   // muestra la métrica de los tipos de letra
   public void paintComponent( Graphics g )
   {
      super.paintComponent( g ); // llama al método paintComponent de la superclase

      g.setFont( new Font( "SansSerif", Font.BOLD, 12 ) );
      FontMetrics metrica = g.getFontMetrics();
      g.drawString( "Tipo de letra actual: " + g.getFont(), 10, 40 );
      g.drawString( "Ascendente: " + metrica.getAscent(), 10, 55 );
      g.drawString( "Descendente: " + metrica.getDescent(), 10, 70 );
      g.drawString( "Altura: " + metrica.getHeight(), 10, 85 );
      g.drawString( "Interlineado: " + metrica.getLeading(), 10, 100 );

      Font tipoLetra = new Font( "Serif", Font.ITALIC, 14 );
      metrica = g.getFontMetrics( tipoLetra );
      g.setFont( tipoLetra );
      g.drawString( "Tipo de letra actual: " + tipoLetra, 10, 130 );
      g.drawString( "Ascendente: " + metrica.getAscent(), 10, 145 );
      g.drawString( "Descendente: " + metrica.getDescent(), 10, 160 );
      g.drawString( "Altura: " + metrica.getHeight(), 10, 175 );
      g.drawString( "Interlineado: " + metrica.getLeading(), 10, 190 );
   } // fin del método paintComponent
} // fin de la clase MetricaJPanel
```

Figura 12.15 | Métrica de los tipos de letra.

```
// Fig. 12.16: Metrica.java
// Muestra la métrica de los tipos de letra.
import javax.swing.JFrame;

public class Metrica
{
   // ejecuta la aplicación
   public static void main( String args[] )
   {
      // crea marco para objeto MetricaJPanel
      JFrame marco = new JFrame( "Demostracion de FontMetrics" );
      marco.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );

      MetricaJPanel metricaJPanel = new MetricaJPanel();
      marco.add( metricaJPanel ); // agrega metricaJPanel al marco
      marco.setSize( 530, 250 ); // establece el tamaño del marco
      marco.setVisible( true ); // muestra el marco
   } // fin de main
} // fin de la clase Metrica
```

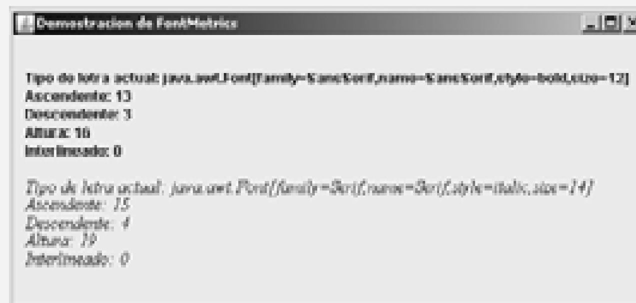


Figura 12.16 | Creación de un objeto `JFrame` para mostrar información sobre la métrica de los tipos de letra.

12.5 Dibujo de líneas, rectángulos y óvalos

En esta sección presentaremos varios métodos de `Graphics` para dibujar líneas, rectángulos y óvalos. Los métodos y sus parámetros se sintetizan en la figura 12.17. Para cada método de dibujo que requiere un parámetro `anchura` y otro `altura`, sus valores deben ser números no negativos. De lo contrario, no se mostrará la figura.

Método	Descripción
`public void drawLine( int x1, int y1, int x2, int y2 )`	
	Dibuja una línea entre el punto (`x1`, `y1`) y el punto (`x2`, `y2`).
`public void drawRect( int x, int y, int anchura, int altura )`	
	Dibuja un rectángulo con la `anchura` y altura especificadas. La esquina superior izquierda del rectángulo tiene las coordenadas (x, y). Sólo el contorno del rectángulo se dibuja usando el color del objeto `Graphics`; el cuerpo del rectángulo no se rellena con este color.
`public void fillRect( int x, int y, int anchura, int altura )`	
	Dibuja un rectángulo relleno con la `anchura` y `altura` especificadas. La esquina superior izquierda del rectángulo tiene las coordenadas (x, y). El rectángulo se rellena con el color del objeto `Graphics`.

Figura 12.17 | Métodos de `Graphics` para dibujar líneas, rectángulos y óvalos. (Parte 1 de 2).

Método	Descripción
`public void clearRect( int x, int y, int anchura, int altura )`	
	Dibuja un rectángulo relleno con la `anchura` y `altura` especificadas, en el color de fondo actual. La esquina superior izquierda del rectángulo tiene las coordenadas (`x`, `y`). Este método es útil si el programador desea eliminar una porción de una imagen.
`public void drawRoundRect( int x, int y, int anchura, int altura, int anchuraArco, int alturaArco )`	
	Dibuja un rectángulo con esquinas redondeadas, en el color actual y con la `anchura` y `altura` especificadas. Los valores de `anchuraArco` y `alturaArco` determinan el grado de redondez de las esquinas (vea la figura 12.20). Sólo se dibuja el contorno de la figura.
`public void fillRoundRect( int x, int y, int anchura, int altura, int anchuraArco, int alturaArco )`	
	Dibuja un rectángulo relleno con esquinas redondeadas, en el color actual y con la `anchura` y `altura` especificadas. Los valores de `anchuraArco` y `alturaArco` determinan el grado de redondez de las esquinas (vea la figura 12.20).
`public void draw3DRect( int x, int y, int anchura, int altura, boolean b )`	
	Dibuja un rectángulo tridimensional en el color actual, con la `anchura` y `altura` especificadas. La esquina superior izquierda del rectángulo tiene las coordenadas (`x`, `y`). El rectángulo aparece con relieve cuando `b` es `true` y sin relieve cuando `b` es `false`. Sólo se dibuja el contorno de la figura.
`public void fill3DRect( int x, int y, int anchura, int altura, boolean b )`	
	Dibuja un rectángulo tridimensional relleno en el color actual, con la `anchura` y `altura` especificadas. La esquina superior izquierda del rectángulo tiene las coordenadas (`x`, `y`). El rectángulo aparece con relieve cuando `b` es `true` y sin relieve cuando `b` es `false`.
`public void drawOval( int x, int y, int anchura, int altura )`	
	Dibuja un óvalo en el color actual, con la `anchura` y `altura` especificadas. La esquina superior izquierda del rectángulo imaginario que lo rodea tiene las coordenadas (`x`, `y`). El óvalo toca los cuatro lados del rectángulo imaginario en el centro de cada uno de los lados (vea la figura 12.21). Sólo se dibuja el contorno de la figura.
`public void fillOval( int x, int y, int anchura, int altura )`	
	Dibuja un óvalo relleno en el color actual, con la `anchura` y `altura` especificadas. La esquina superior izquierda del rectángulo imaginario que lo rodea tiene las coordenadas (`x`, `y`). El óvalo toca los cuatro lados del rectángulo imaginario en el centro de cada uno de los lados (vea la figura 12.21).

Figura 12.17 | Métodos de `Graphics` para dibujar líneas, rectángulos y óvalos. (Parte 2 de 2).

La aplicación de las figuras 12.18 y 12.19 demuestra cómo dibujar una variedad de líneas, rectángulos, rectángulos tridimensionales, rectángulos con esquinas redondeadas y óvalos.

En la figura 12.18, en la línea 17 se dibuja una línea roja, en la línea 20 se dibuja un rectángulo vacío de color azul y en la línea 21 se dibuja un rectángulo relleno de color azul. Los métodos **`fillRoundRect`** (línea 24) y **`drawRoundRect`** (línea 25) dibujan rectángulos con esquinas redondeadas. Sus primeros dos argumentos especifican las coordenadas de la esquina superior izquierda del **rectángulo delimitador** (el área en la que se dibujará el rectángulo redondeado). Observe que las coordenadas de la esquina superior izquierda no son el borde del rectángulo redondeado, sino las coordenadas en donde se encontraría el borde si el rectángulo tuviera esquinas cuadradas. Los argumentos tercero y cuarto especifican la anchura y altura del rectángulo. Sus últimos dos argumentos determinan los diámetros vertical y horizontal del arco (es decir, la anchura y la altura del arco) que se utiliza para representar las esquinas.

```
// Fig. 12.18: LineasRectsOvalosJPanel.java
// Dibujo de líneas, rectángulos y óvalos.
import java.awt.Color;
import java.awt.Graphics;
import javax.swing.JPanel;

public class LineasRectsOvalosJPanel extends JPanel
{
   // muestra varias líneas, rectángulos y óvalos
   public void paintComponent( Graphics g )
   {
      super.paintComponent( g ); // llama al método paintComponent de la superclase

      this.setBackground( Color.WHITE );

      g.setColor( Color.RED );
      g.drawLine( 5, 30, 380, 30 );

      g.setColor( Color.BLUE );
      g.drawRect( 5, 40, 90, 55 );
      g.fillRect( 100, 40, 90, 55 );

      g.setColor( Color.CYAN );
      g.fillRoundRect( 195, 40, 90, 55, 50, 50 );
      g.drawRoundRect( 290, 40, 90, 55, 20, 20 );

      g.setColor( Color.YELLOW );
      g.draw3DRect( 5, 100, 90, 55, true );
      g.fill3DRect( 100, 100, 90, 55, false );

      g.setColor( Color.MAGENTA );
      g.drawOval( 195, 100, 90, 55 );
      g.fillOval( 290, 100, 90, 55 );
   } // fin del método paintComponent
} // fin de la clase LineasRectsOvalosJPanel
```

Figura 12.18 | Dibujo de líneas, rectángulos y óvalos.

```
// Fig. 12.19: LineasRectsOvalos.java
// Dibujo de líneas, rectángulos y óvalos.
import java.awt.Color;
import javax.swing.JFrame;

public class LineasRectsOvalos
{
   // ejecuta la aplicación
   public static void main( String args[] )
   {
      // crea marco para LineasRectsOvalosJPanel
      JFrame marco =
         new JFrame( "Dibujo de lineas, rectangulos y ovalos" );
      marco.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );

      LineasRectsOvalosJPanel lineasRectsOvalosJPanel =
         new LineasRectsOvalosJPanel();
      lineasRectsOvalosJPanel.setBackground( Color.WHITE );
      marco.add( lineasRectsOvalosJPanel ); // agrega el panel al marco
```

Figura 12.19 | Creación de `JFrame` para mostrar líneas, rectángulos y óvalos. (Parte 1 de 2).

```
        marco.setSize( 400, 210 ); // establece el tamaño del marco
        marco.setVisible( true ); // muestra el marco
    } // fin de main
} // fin de la clase LineasRectsOvalos
```

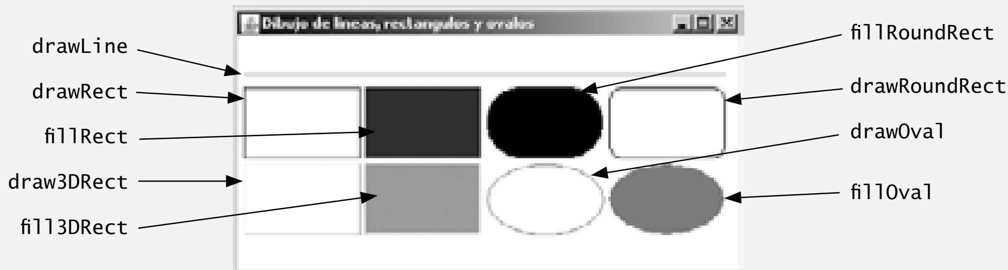


Figura 12.19 | Creación de `JFrame` para mostrar líneas, rectángulos y óvalos. (Parte 2 de 2).

En la figura 12.20 se muestran la anchura y altura del arco, junto con la anchura y la altura de un rectángulo redondeado. Si se utiliza el mismo valor para la anchura y la altura del arco, se produce un cuarto de círculo en cada esquina. Cuando la anchura y la altura del arco, la anchura y la altura del rectángulo tienen los mismos valores, el resultado es un círculo. Si los valores para `anchura` y `altura` son los mismos, y los valores de `anchuraArco` y `alturaArco` son `0`, el resultado es un cuadrado.

Los métodos **`draw3DRect`** (línea 28) y **`fill3DRect`** (línea 29) reciben los mismos argumentos. Los primeros dos argumentos especifican la esquina superior izquierda del rectángulo. Los siguientes dos argumentos especifican la anchura y altura del rectángulo, respectivamente. El último argumento determina si el rectángulo está **con relieve** (true) o **sin relieve** (false). El efecto tridimensional de `draw3DRect` aparece como dos bordes del rectángulo en el color original y dos bordes en un color ligeramente más oscuro. El efecto tridimensional de `fill3DRect` aparece como dos bordes del rectángulo en el color del dibujo original y los otros dos bordes y el relleno en un color ligeramente más oscuro. Los rectángulos con relieve tienen los bordes de color original del dibujo en las partes superior e izquierda del rectángulo. Los rectángulos sin relieve tienen los bordes de color original del dibujo en las partes inferior y derecha del rectángulo. El efecto tridimensional es difícil de ver en ciertos colores.

Los métodos **`drawOval`** y **`fillOval`** (líneas 32 y 33) reciben los mismos cuatro argumentos. Los primeros dos argumentos especifican la coordenada superior izquierda del rectángulo delimitador que contiene el óvalo. Los últimos dos argumentos especifican la anchura y la altura del rectángulo delimitador, respectivamente. En la figura 12.21 se muestra un óvalo delimitado por un rectángulo. Observe que el óvalo toca el centro de los cuatro lados del rectángulo delimitador. (El rectángulo delimitador no se muestra en la pantalla).

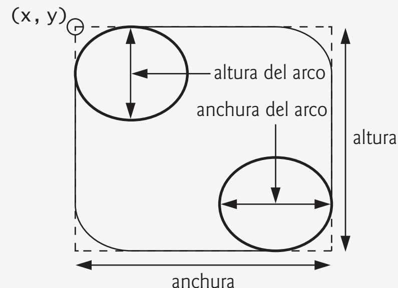


Figura 12.20 | Anchura y altura del arco para los rectángulos redondeados.

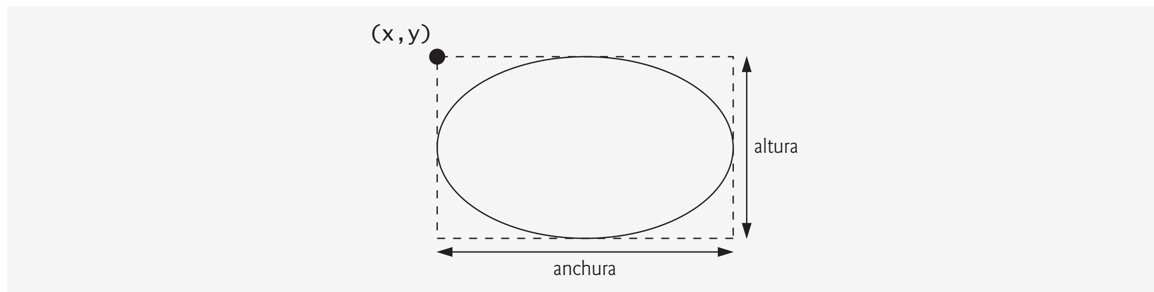


Figura 12.21 | Óvalo delimitado por un rectángulo.

12.6 Dibujo de arcos

Un **arco** se dibuja como una porción de un óvalo. Los ángulos de los arcos se miden en grados. Los arcos se **extienden** (es decir, se mueven a lo largo de una curva) desde un **ángulo inicial**, en base al número de grados especificados por el **ángulo del arco**. El ángulo inicial indica, en grados, en dónde empieza el arco. El ángulo del arco especifica el número total de grados hasta los que se va a extender el arco. En la figura 12.22 se muestran dos arcos. El conjunto izquierdo de ejes muestra a un arco extendiéndose desde cero hasta aproximadamente 110 grados. Los arcos que se extienden en dirección en contra de las manecillas del reloj se miden en **grados positivos**. El conjunto derecho de ejes muestra a un arco extendiéndose desde cero hasta aproximadamente –110 grados. Los arcos que se extienden en dirección a favor de las manecillas del reloj se miden en **grados negativos**. Observe los cuadros punteados alrededor de los arcos en la figura 12.22. Cuando dibujamos un arco, debemos especificar un rectángulo delimitador para un óvalo. El arco se extenderá a lo largo de una parte del óvalo. Los métodos **`drawArc`** y **`fillArc`** de `Graphics` para dibujar arcos se sintetizan en la figura 12.23.

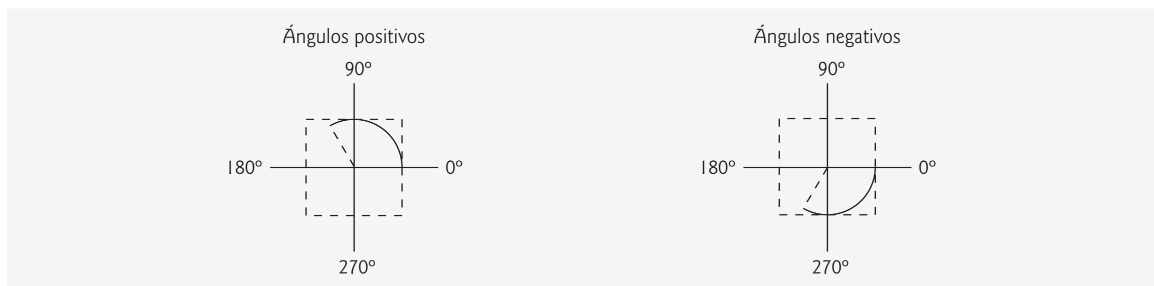


Figura 12.22 | Ángulos positivos y negativos de un arco.

Método	Descripción
`public void drawArc( int x, int y, int anchura, int altura, int anguloInicial, int anguloArco )`	
	Dibuja un arco relativo a las coordenadas (`x`, `y`) de la esquina superior izquierda del rectángulo delimitador, con la `anchura` y `altura` especificadas. El segmento del arco se dibuja empezando en `anguloInicial` y se extiende hasta los grados especificados por `anguloArco`.
`public void fillArc( int x, int y, int anchura, int altura, int anguloInicial, int anguloArco )`	
	Dibuja un arco relleno (es decir, un sector) relativo a las coordenadas (`x`, `y`) de la esquina superior izquierda del rectángulo delimitador, con la `anchura` y `altura` especificadas. El segmento del arco se dibuja empezando en `anguloInicial` y se extiende hasta los grados especificados por `anguloArco`.

Figura 12.23 | Métodos de `Graphics` para dibujar arcos.

La aplicación de las figuras 12.24 y 12.25 demuestra el uso de los métodos para arcos de la figura 12.23. La aplicación dibuja seis arcos (tres sin rellenar y tres rellenos). Para ilustrar el rectángulo delimitador que ayuda a determinar en dónde aparece el arco, los primeros tres arcos se muestran dentro de un rectángulo amarillo que tiene los mismos argumentos `x`, `y`, `anchura` y `altura` que los arcos.

```
// Fig. 12.24: ArcosJPanel.java
// Dibujo de arcos.
import java.awt.Color;
import java.awt.Graphics;
import javax.swing.JPanel;

public class ArcosJPanel extends JPanel
{
   // dibuja rectángulos y arcos
   public void paintComponent( Graphics g )
   {
      super.paintComponent( g ); // llama al método paintComponent de la superclase

      // empieza en 0 y se extiende hasta 360 grados
      g.setColor( Color.RED );
      g.drawRect( 15, 35, 80, 80 );
      g.setColor( Color.BLACK );
      g.drawArc( 15, 35, 80, 80, 0, 360 );

      // empieza en 0 y se extiende hasta 110
      g.setColor( Color.RED );
      g.drawRect( 100, 35, 80, 80 );
      g.setColor( Color.BLACK );
      g.drawArc( 100, 35, 80, 80, 0, 110 );

      // empieza en 0 y se extiende hasta -270 grados
      g.setColor( Color.RED );
      g.drawRect( 185, 35, 80, 80 );
      g.setColor( Color.BLACK );
      g.drawArc( 185, 35, 80, 80, 0, -270 );

      // empieza en 0 y se extiende hasta 360 grados
      g.fillArc( 15, 120, 80, 40, 0, 360 );

      // empieza en 270 y se extiende hasta -90 grados
      g.fillArc( 100, 120, 80, 40, 270, -90 );

      // empieza en 0 y se extiende hasta -270 grados
      g.fillArc( 185, 120, 80, 40, 0, -270 );
   } // fin del método paintComponent
} // fin de la clase ArcosJPanel
```

Figura 12.24 | Arcos mostrados con `drawArc` y `fillArc`.

```
// Fig. 12.25: DibujarArcos.java
// Dibujo de arcos.
import javax.swing.JFrame;

public class DibujarArcos
{
```

Figura 12.25 | Creación de un objeto `JFrame` para mostrar arcos. (Parte 1 de 2).

```
    // ejecuta la aplicación
    public static void main( String args[] )
    {
       // crea marco para ArcosJPanel
       JFrame marco = new JFrame( "Dibujo de arcos" );
       marco.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );

       ArcosJPanel arcosJPanel = new ArcosJPanel(); // crea objeto ArcosJPanel
       marco.add( arcosJPanel ); // agrega arcosJPanel al marco
       marco.setSize( 300, 210 ); // establece el tamaño del marco
       marco.setVisible( true ); // muestra el marco
    } // fin de main
 } // fin de la clase DibujarArcos
```

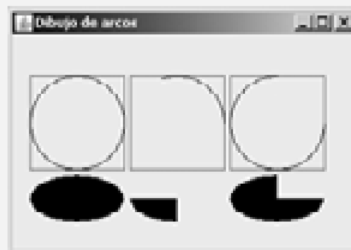


Figura 12.25 | Creación de un objeto `JFrame` para mostrar arcos. (Parte 2 de 2).

12.7 Dibujo de polígonos y polilíneas

Los **polígonos** son figuras cerradas de varios lados, compuestas por segmentos de línea recta. Las **polilíneas** son una secuencia de puntos conectados. En la figura 12.26 describimos los métodos para dibujar polígonos y polilíneas. Observe que algunos métodos requieren un objeto **`Polygon`** (paquete `java.awt`). Los constructores de la clase `Polygon` se describen también en la figura 12.26. La aplicación de las figuras 12.27 y 12.28 dibuja polígonos y polilíneas.

Método	Descripción
Métodos de `Graphics` para dibujar polígonos	
`public void drawPolygon( int puntosX[], int puntosY[], int puntos )`	
	Dibuja un polígono. La coordenada *x* de cada punto se especifica en el arreglo `puntosX` y la coordenada *y* de cada punto se especifica en el arreglo `puntosY`. El último argumento especifica el número de `puntos`. Este método dibuja un polígono cerrado. Si el último punto es distinto del primero, el polígono se cierra mediante una línea que conecte el último punto con el primero.
`public void drawPolyline( int puntosX[], int puntosY[], int puntos )`	
	Dibuja una secuencia de líneas conectadas. La coordenada *x* de cada punto se especifica en el arreglo `puntosX` y la coordenada *y* de cada punto se especifica en el arreglo `puntosY`. El último argumento especifica el número de `puntos`. Si el último punto es distinto del primero, la polilínea no se cierra.

Figura 12.26 | Métodos de `Graphics` para dibujar polígonos y métodos de la clase `Polygon`. (Parte 1 de 2).

Método	Descripción
`public void drawPolygon( Polygon p )`	Dibuja el polígono especificado.
`public void fillPolygon( int puntosX[], int puntosY[], int puntos )`	Dibuja un polígono relleno. La coordenada *x* de cada punto se especifica en el arreglo `puntosX` y la coordenada *y* de cada punto se especifica en el arreglo `puntosY`. El último argumento especifica el número de `puntos`. Este método dibuja un polígono cerrado. Si el último punto es distinto del primero, el polígono se cierra mediante una línea que conecte el último punto con el primero.
`public void fillPolygon( Polygon p )`	Dibuja el polígono relleno especificado. El polígono es cerrado.
Constructores y métodos de `Polygon`	
`public Polygon()`	Crea un nuevo objeto polígono. Este objeto no contiene ningún punto.
`public Polygon( int valoresX[], int valoresY[], int numeroDePuntos )`	Crea un nuevo objeto polígono. Este objeto tiene `numeroDePuntos` lados, en donde cada punto consiste de una coordenada *x* desde `valoresX`, y una coordenada *y* desde `valoresY`.
`public void addPoint( int x, int y )`	Agrega pares de coordenadas *x* y *y* al objeto `Polygon`.

Figura 12.26 | Métodos de `Graphics` para dibujar polígonos y métodos de la clase `Polygon`. (Parte 2 de 2).

```
// Fig. 12.27: PoligonosJPanel.java
// Dibujo de polígonos.
import java.awt.Graphics;
import java.awt.Polygon;
import javax.swing.JPanel;

public class PoligonosJPanel extends JPanel
{
   // dibuja polígonos y polilíneas
   public void paintComponent( Graphics g )
   {
      super.paintComponent( g ); // llama al método paintComponent de la superclase

      // dibuja polígono con objeto polígono
      int valoresX[] = { 20, 40, 50, 30, 20, 15 };
      int valoresY[] = { 50, 50, 60, 80, 80, 60 };
      Polygon poligono1 = new Polygon( valoresX, valoresY, 6 );
      g.drawPolygon( poligono1 );

      // dibuja polilíneas con dos arreglos
      int valoresX2[] = { 70, 90, 100, 80, 70, 65, 60 };
```

Figura 12.27 | Polígonos mostrados con `drawPolygon` y `fillPolygon`. (Parte 1 de 2).

```
22         int valoresY2[] = { 100, 100, 110, 110, 130, 110, 90 };
23         g.drawPolyline( valoresX2, valoresY2, 7 );
24
25         // rellena polígono con dos arreglos
26         int valoresX3[] = { 120, 140, 150, 190 };
27         int valoresY3[] = { 40, 70, 80, 60 };
28         g.fillPolygon( valoresX3, valoresY3, 4 );
29
30         // dibuja polígono relleno con objeto Polygon
31         Polygon poligono2= new Polygon();
32         poligono2.addPoint( 165, 135 );
33         poligono2.addPoint( 175, 150 );
34         poligono2.addPoint( 270, 200 );
35         poligono2.addPoint( 200, 220 );
36         poligono2.addPoint( 130, 180 );
37         g.fillPolygon( poligono2 );
38      } // fin del método paintComponent
39   } // fin de la clase PoligonosJPanel
```

Figura 12.27 | Polígonos mostrados con `drawPolygon` y `fillPolygon`. (Parte 2 de 2).

```
// Fig. 12.28: DibujarPoligonos.java
// Dibujo de polígonos.
import javax.swing.JFrame;

public class DibujarPoligonos
{
   // ejecuta la aplicación
   public static void main( String args[] )
   {
      // crea marco para objeto PoligonosJPanel
      JFrame marco = new JFrame( "Dibujo de poligonos" );
      marco.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );

      PoligonosJPanel poligonosJPanel = new PoligonosJPanel();
      marco.add( poligonosJPanel ); // agrega poligonosJPanel al marco
      marco.setSize( 280, 270 ); // establece el tamaño del marco
      marco.setVisible( true ); // muestra el marco
   } // fin de main
} // fin de la clase DibujarPoligonos
```

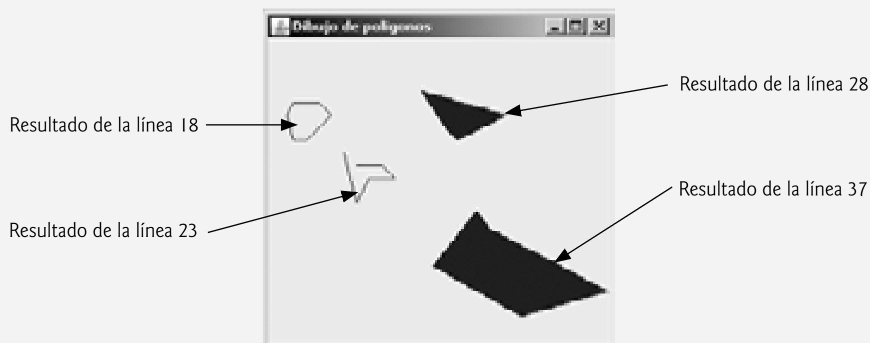


Figura 12.28 | Creación de un objeto `JFrame` para mostrar polígonos.

En las líneas 15 y 16 de la figura 12.27 se crean dos arreglos `int` y se utilizan para especificar los puntos del objeto `Polygon` llamado `poligono1`. La llamada al constructor de `Polygon` en la línea 17 recibe el arreglo `valoresX`, el cual contiene la coordenada *x* de cada punto; el arreglo `valoresY`, que contiene la coordenada *y* de cada punto y el número 6 (el número de puntos en el polígono). En la línea 18 se muestra `poligono1` al pasarlo como argumento para el método **`drawPolygon`** de `Graphics`.

En las líneas 21 y 22 se crean dos arreglos `int` y se utilizan para especificar los puntos de una serie de líneas conectadas. El arreglo `valoresX2` contiene la coordenada *x* de cada punto y el arreglo `valoresY2` contiene la coordenada *y* de cada punto. En la línea 23 se utiliza el método **`drawPolyline`** de `Graphics` para mostrar la serie de líneas conectadas que se especifican mediante los argumentos `valoresX2`, `valoresY2` y `7` (el número de puntos).

En las líneas 26 y 27 se crean dos arreglos `int` y se utilizan para especificar los puntos de un polígono. El arreglo `valoresX3` contiene la coordenada *x* de cada punto y el arreglo `valoresY3` contiene la coordenada *y* de cada punto. En la línea 28 se muestra un polígono al pasar al método **`fillPolygon`** de `Graphics` los dos arreglos (`valoresX3` y `valoresY3`) y el número de puntos a dibujar (`4`).



Error común de programación 12.1

Se lanzará una excepción `ArrayIndexOutOfBoundsException` si el número de puntos especificados en el tercer argumento del método `drawPolygon` o del método `fillPolygon` es mayor que el número de elementos en los arreglos de las coordenadas que especifican el polígono a mostrar.

En la línea 31 se crea el objeto `Polygon` llamado `poligono2`, sin puntos. En las líneas 32 a 36 se utiliza el método **`addPoint`** de `Polygon` para agregar pares de coordenadas *x* y *y* al objeto `Polygon`. En la línea 37 se muestra el objeto `Polygon` llamado `poligono2`, al pasarlo al método `fillPolygon` de `Graphics`.

12.8 La API Java 2D

La **API Java 2D** proporciona herramientas avanzadas para gráficos bidimensionales, para los programadores que requieren manipulaciones gráficas detalladas y complejas. La API incluye características para procesar arte lineal, texto e imágenes en los paquetes `java.awt`, `java.awt.image`, `java.awt.color`, `java.awt.font`, `java.awt.geom`, `java.awt.print` y `java.awt.image.renderable`. Las herramientas de la API son muy extensas como para cubrirlas todas en este libro. Para ver las generalidades acerca de estas herramientas, consulte la demostración de Java 2D (que veremos en el capítulo 20, Introducción a las applets de Java) o visite la página Web `java.sun.com/products/java-media/2D/index.html`. En esta sección veremos las generalidades de varias herramientas de Java 2D.

El dibujo con la API Java 2D se logra mediante el uso de una referencia **`Graphics2D`** (paquete `java.awt`), que es una subclase abstracta de la clase `Graphics`, por lo que tiene todas las herramientas para gráficos que se demostraron anteriormente en este capítulo. De hecho, el objeto en sí utilizado para dibujar en todos los métodos `paintComponent` es una instancia de una subclase de `Graphics2D` que se pasa al método `paintComponent` y se utiliza mediante la superclase `Graphics`. Para acceder a las herramientas de `Graphics2D`, debemos convertir la referencia `Graphics` (`g`) que se pasa a `paintComponent` en una referencia `Graphics2D`, mediante una instrucción como:

```
Graphics2D g2d = ( Graphics2D ) g;
```

Los siguientes dos ejemplos utilizan esta técnica.

Líneas, rectángulos, rectángulos redondeados, arcos y elipses

En el siguiente ejemplo se muestran varias figuras de Java 2D del paquete `java.awt.geom`, incluyendo a **`Line2D.Double`**, **`Rectangle2D.Double`**, **`RoundRectangle2D.Double`**, **`Arc2D.Double`** y **`Ellipse2D.Double`**. Observe la sintaxis de cada uno de los nombres de las clases. Cada una de estas clases representa una figura con las dimensiones especificadas como valores de punto flotante con doble precisión. Hay una versión separada de cada figura, representada con valores de punto flotante con precisión simple (como **`Ellipse2D.Float`**). En cada caso, `Double` es una clase `static` anidada de la clase que se especifica a la izquierda del punto (por ejemplo, `Ellipse2D`). Para utilizar la clase `static` anidada, simplemente debemos calificar su nombre con el nombre de la clase externa.

En las figuras 12.29 y 12.30, dibujamos figuras de Java 2D y modificamos sus características de dibujo, como cambiar el grosor de línea, rellenar figuras con patrones y dibujar líneas punteadas. Éstas son sólo algunas de las muchas herramientas que proporciona Java 2D.

```
// Fig. 12.29: FigurasJPanel.java
// Demostración de algunas figuras de Java 2D.
import java.awt.Color;
import java.awt.Graphics;
import java.awt.BasicStroke;
import java.awt.GradientPaint;
import java.awt.TexturePaint;
import java.awt.Rectangle;
import java.awt.Graphics2D;
import java.awt.geom.Ellipse2D;
import java.awt.geom.Rectangle2D;
import java.awt.geom.RoundRectangle2D;
import java.awt.geom.Arc2D;
import java.awt.geom.Line2D;
import java.awt.image.BufferedImage;
import javax.swing.JPanel;

public class FigurasJPanel extends JPanel
{
   // dibuja figuras con la API Java 2D
   public void paintComponent( Graphics g )
   {
      super.paintComponent( g ); // llama al método paintComponent de la superclase

      Graphics2D g2d = ( Graphics2D ) g; // convierte a g en objeto Graphics2D

      // dibuja un elipse en 2D, relleno con un gradiente color azul-amarillo
      g2d.setPaint( new GradientPaint( 5, 30, Color.BLUE, 35, 100,
         Color.YELLOW, true ) );
      g2d.fill( new Ellipse2D.Double( 5, 30, 65, 100 ) );

      // dibuja rectángulo en 2D de color rojo
      g2d.setPaint( Color.RED );
      g2d.setStroke( new BasicStroke( 10.0f ) );
      g2d.draw( new Rectangle2D.Double( 80, 30, 65, 100 ) );

      // dibuja rectángulo delimitador en 2D, con un fondo con búfer
      BufferedImage imagenBuf = new BufferedImage( 10, 10,
         BufferedImage.TYPE_INT_RGB );

      // obtiene objeto Graphics2D de imagenBuf y dibuja en él
      Graphics2D gg = imagenBuf.createGraphics();
      gg.setColor( Color.YELLOW ); // dibuja en color amarillo
      gg.fillRect( 0, 0, 10, 10 ); // dibuja un rectángulo relleno
      gg.setColor( Color.BLACK );  // dibuja en color negro
      gg.drawRect( 1, 1, 6, 6 ); // dibuja un rectángulo
      gg.setColor( Color.BLUE ); // dibuja en color azul
      gg.fillRect( 1, 1, 3, 3 ); // dibuja un rectángulo relleno
      gg.setColor( Color.RED ); // dibuja en color rojo
      gg.fillRect( 4, 4, 3, 3 ); // dibuja un rectángulo relleno

      // pinta a imagenBuf en el objeto JFrame
      g2d.setPaint( new TexturePaint( imagenBuf,
         new Rectangle( 10, 10 ) ) );
      g2d.fill(
         new RoundRectangle2D.Double( 155, 30, 75, 100, 50, 50 ) );

      // dibuja arco en forma de pastel en 2D, de color blanco
      g2d.setPaint( Color.WHITE );
```

Figura 12.29 | Figuras de Java 2D. (Parte 1 de 2).

```
60         g2d.setStroke( new BasicStroke( 6.0f ) );
61         g2d.draw(
62            new Arc2D.Double( 240, 30, 75, 100, 0, 270, Arc2D.PIE ) );
63
64         // dibuja líneas 2D en verde y amarillo
65         g2d.setPaint( Color.GREEN );
66         g2d.draw( new Line2D.Double( 395, 30, 320, 150 ) );
67
68         // dibuja línea 2D usando el trazo
69         float guiones[] = { 10 }; // especifica el patrón de guiones
70         g2d.setPaint( Color.YELLOW );
71         g2d.setStroke( new BasicStroke( 4, BasicStroke.CAP_ROUND,
72            BasicStroke.JOIN_ROUND, 10, guiones, 0 ) );
73         g2d.draw( new Line2D.Double( 320, 30, 395, 150 ) );
74      } // fin del método paintComponent
75   } // fin de la clase FigurasJPanel
```

Figura 12.29 | Figuras de Java 2D. (Parte 2 de 2).

```
// Fig. 12.30: Figuras.java
// Demostración de algunas figuras de Java 2D.
import javax.swing.JFrame;

public class Figuras
{
   // ejecuta la aplicación
   public static void main( String args[] )
   {
      // crea marco para objeto FigurasJPanel
      JFrame marco = new JFrame( "Dibujo de figuras en 2D" );
      marco.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );

      // crea objeto FigurasJPanel
      FigurasJPanel figurasJPanel = new FigurasJPanel();

      marco.add( figurasJPanel ); // agrega figurasJPanel to marco
      marco.setSize( 425, 200 ); // establece el tamaño del marco
      marco.setVisible( true ); // muestra el marco
   } // fin de main
} // fin de la clase Figuras
```

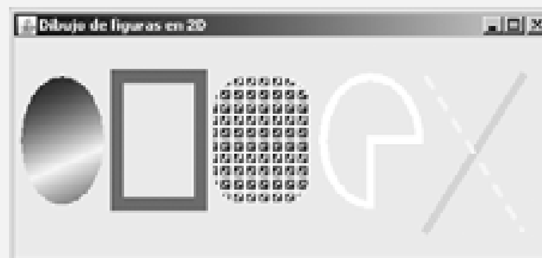


Figura 12.30 | Creación de un objeto `JFrame` para mostrar figuras.

En la línea 25 de la figura 12.29 se convierte la referencia `Graphics` recibida por `paintComponent` a una referencia `Graphics2D`, y se asigna a `g2d` para permitir el acceso a las características de Java 2D.

Óvalos, rellenos con degradado y objetos `Paint`

La primera figura que dibujamos es un óvalo relleno con colores que cambian gradualmente. En las líneas 28 y 29 se invoca el método **setPaint** de Graphics2D para establecer el objeto **Paint** que determina el color para la figura a mostrar. Un objeto Paint implementa a la interfaz java.awt.Paint. Puede ser algo tan simple como uno de los objetos Color previamente declarados, los cuales se presentaron en la sección 12.3 (la clase Color implementa a Paint), o el objeto Paint puede ser una instancia de las clases GradientPaint, **SystemColor**, TexturePaint, LinearGradientPain o RadientGradientPaint de la API Java2D. En este caso, utilizamos un objeto GradientPaint.

La clase GradientPaint ayuda a dibujar una figura en colores que cambian gradualmente (lo cual se conoce como **degradado**). El constructor de GradientPaint que se utiliza aquí requiere siete argumentos. Los primeros dos especifican la coordenada inicial del degradado. El tercer argumento especifica el Color inicial del degradado. Los argumentos cuarto y quinto especifican la coordenada final del degradado. El sexto especifica el Color final del degradado y el último especifica si el degradado es **cíclico** (true) o **acíclico** (false). Los dos conjuntos de coordenadas determinan la dirección del degradado. Como la segunda coordenada (35, 100) se encuentra hacia abajo y a la derecha de la primera coordenada (5, 30), el degradado va hacia abajo y a la derecha con cierto ángulo. Como este degradado es cíclico (true), el color empieza con azul, se convierte gradualmente en amarillo y luego regresa gradualmente a azul. Si el degradado es acíclico, el color cambia del primer color especificado (por ejemplo, azul) al segundo color (por ejemplo, amarillo).

En la línea 30 se utiliza el método **fill** de Graphics2D para dibujar un objeto **Shape** relleno (un objeto que implementa a la interfaz Shape del paquete java.awt). En este caso mostramos un objeto Ellipse2D.Double. El constructor de Ellipse2D.Double recibe cuatro argumentos que especifican el rectángulo delimitador para mostrar la elipse.

Rectángulos, trazos (objetos `Stroke`*)*

A continuación dibujamos un rectángulo rojo con un borde grueso. En la línea 33 se utiliza setPaint para establecer el objeto Paint en Color.RED. En la línea 34 se utiliza el método **setStroke** de Graphics2D para establecer las características del borde del rectángulo (o las líneas para cualquier otra figura). El método setStroke requiere como argumento un objeto que implemente a la interfaz **Stroke** (paquete java.awt). En este caso, utilizamos una instancia de la clase BasicStroke. Esta clase proporciona varios constructores para especificar la anchura de la línea, la manera en que ésta termina (lo cual se le conoce como **cofias**), la manera en que las líneas se unen entre sí (lo cual se le conoce como **uniones de línea**) y los atributos de los guiones de la línea (si es una línea punteada). El constructor aquí especifica que la línea debe tener una anchura de 10 píxeles.

En la línea 35 se utiliza el método **draw** de Graphics2D para dibujar un objeto Shape; en este caso, una instancia de la clase Rectangle2D.Double. El constructor de Rectangle2D.Double recibe cuatro argumentos que especifican las coordenadas *x* y *y* de la esquina superior izquierda, la anchura y la altura del rectángulo.

Rectángulos redondeados, objetos `BufferedImage` *y* `TexturePaint`

A continuación dibujamos un rectángulo redondeado, relleno con un patrón creado en un objeto **BufferedImage** (paquete java.awt.image). En las líneas 38 y 39 se crea el objeto BufferedImage. La clase BufferedImage puede usarse para producir imágenes en color y escala de grises. Este objeto BufferedImage en particular tiene una anchura y una altura de 10 píxeles (según lo especificado por los primeros dos argumentos del constructor). El tercer argumento del constructor, **BufferedImage.TYPE_INT_RGB**, indica que la imagen se almacena en color, utilizando el esquema de colores RGB.

Para crear el patrón de relleno para el rectángulo redondeado, debemos primero dibujar en el objeto BufferedImage. En la línea 42 se crea un objeto Graphics2D (con una llamada al método **createGraphics** de BufferedImage) que puede usarse para dibujar en el objeto BufferedImage. En las líneas 43 a 50 se utilizan los métodos setColor, fillRect y drawRect (descritos anteriormente en este capítulo) para crear el patrón.

En las líneas 53 y 54 se establece el objeto Paint en un nuevo objeto TexturePaint (paquete java.awt). Un objeto TexturePaint utiliza la imagen almacenada en su objeto BufferedImage asociado (el primer argumento del constructor) como la textura para rellenar una figura. El segundo argumento especifica el área Rectangle del objeto BufferedImage que se repetirá en toda la textura. En este caso, el objeto Rectangle es del mismo tamaño que el objeto BufferedImage. Sin embargo, puede utilizarse una porción más pequeña del objeto BufferedImage.

En las líneas 55 y 56 se utiliza el método `fill` de `Graphics2D` para dibujar un objeto `Shape` relleno; en este caso, una instancia de la clase `RoundRectangle2D.Double`. El constructor de la clase `RoundRectangle2D.Double` recibe seis argumentos que especifican las dimensiones del rectángulo, la anchura y la altura del arco utilizado para redondear las esquinas.

Arcos

A continuación dibujamos un arco en forma de pastel, con una línea blanca gruesa. En la línea 59 se establece el objeto `Paint` en `Color.WHITE`. En la línea 60 se establece el objeto `Stroke` en un nuevo objeto `BasicStroke` para una línea con 6 píxeles de anchura. En las líneas 61 y 62 se utiliza el método `draw` de `Graphics2D` para dibujar un objeto `Shape`; en este caso, un `Arc2D.Double`. Los primeros cuatro argumentos del constructor de `Arc2D.Double` especifican las coordenadas *x* y *y* de la esquina superior izquierda, la anchura y la altura del rectángulo delimitador para el arco. El quinto argumento especifica el ángulo inicial. El sexto especifica el ángulo del arco. El último argumento especifica cómo se cierra el arco. La constante **`Arc2D.PIE`** indica que el arco se cierra dibujando dos líneas: una línea va desde el punto inicial del arco hasta el centro del rectángulo delimitador, y otra va desde el centro del rectángulo delimitador hasta el punto final. La clase `Arc2D` proporciona otras dos constantes estáticas para especificar cómo se cierra el arco. La constante **`Arc2D.CHORD`** dibuja una línea que va desde el punto inicial hasta el punto final. La constante **`Arc2D.OPEN`** especifica que el arco no debe cerrarse.

Líneas

Finalmente, dibujamos dos líneas utilizando objetos **`Line2D`**: una sólida y una punteada. En la línea 65 se establece el objeto `Paint` en `Color.GREEN`. En la línea 66 se utiliza el método `draw` de `Graphics2D` para dibujar un objeto `Shape`; en este caso, una instancia de la clase `Line2D.Double`. Los argumentos del constructor de `Line2D.Double` especifican las coordenadas inicial y final de la línea.

En la línea 69 se declara un arreglo `float` de un elemento, el cual contiene el valor 10. Este arreglo debe utilizarse para describir los guiones en la línea punteada. En este caso, cada guión será de 10 píxeles de largo. Para crear guiones de diferentes longitudes en un patrón, simplemente debe proporcionar la longitud de cada guión como un elemento en el arreglo. En la línea 70 se establece el objeto `Paint` en `Color.YELLOW`. En las líneas 71 y 72 se establece el objeto `Stroke` en un nuevo objeto `BasicStroke`. La línea tendrá una anchura de 4 píxeles y extremos redondeados (**`BasicStroke.CAP_ROUND`**). Si las líneas se unen entre sí (como en un rectángulo en las esquinas), la unión de las líneas será redondeada (**`BasicStroke.JOIN_ROUND`**). El argumento `guiones` especifica las longitudes de los guiones de la línea. El último argumento indica el índice inicial en el arreglo guiones para el primer guión en el patrón. En la línea 73 se dibuja una línea con el objeto `Stroke` actual.

Creación de sus propias figuras mediante las rutas generales

A continuación presentaremos una **`ruta general`**: una figura compuesta de líneas rectas y curvas complejas. Una ruta general se representa con un objeto de la clase **`GeneralPath`** (paquete `java.awt.geom`). La aplicación de las figuras 12.31 y 12.32 demuestra cómo dibujar una ruta general, en forma de una estrella de cinco puntas.

```java
// Fig. 12.31: Figuras2JPanel.java
// Demostración de una ruta general.
import java.awt.Color;
import java.awt.Graphics;
import java.awt.Graphics2D;
import java.awt.geom.GeneralPath;
import java.util.Random;
import javax.swing.JPanel;

public class Figuras2JPanel extends JPanel
{
   // dibuja rutas generales
   public void paintComponent( Graphics g )
```

Figura 12.31 | Rutas generales de Java 2D. (Parte 1 de 2).

```
    {
       super.paintComponent( g ); // llama al método paintComponent de la superclase
       Random aleatorio = new Random(); // obtiene el generador de números aleatorios

       int puntosX[] = { 55, 67, 109, 73, 83, 55, 27, 37, 1, 43 };
       int puntosY[] = { 0, 36, 36, 54, 96, 72, 96, 54, 36, 36 };

       Graphics2D g2d = ( Graphics2D ) g;
       GeneralPath estrella = new GeneralPath(); // crea objeto GeneralPath

       // establece la coordenada inicial de la ruta general
       estrella.moveTo( puntosX[ 0 ], puntosY[ 0 ] );

       // crea la estrella; esto no la dibuja
       for ( int cuenta = 1; cuenta < puntosX.length; cuenta++ )
          estrella.lineTo( puntosX[ cuenta ], puntosY[ cuenta ] );

       estrella.closePath(); // cierra la figura

       g2d.translate( 200, 200 ); // traslada el origen a (200, 200)

       // gira alrededor del origen y dibuja estrellas en colores aleatorios
       for ( int cuenta = 1; cuenta <= 20; cuenta++ )
       {
          g2d.rotate( Math.PI / 10.0 ); // gira el sistema de coordenadas

          // establece el color de dibujo al azar
          g2d.setColor( new Color( aleatorio.nextInt( 256 ),
             aleatorio.nextInt( 256 ), aleatorio.nextInt( 256 ) ) );

          g2d.fill( estrella ); // dibuja estrella rellena
       } // fin de for
    } // fin del método paintComponent
} // fin de la clase Figuras2JPanel
```

Figura 12.31 | Rutas generales de Java 2D. (Parte 2 de 2).

```
// Fig. 12.32: Figuras2.java
// Demostración de una ruta general.
import java.awt.Color;
import javax.swing.JFrame;

public class Figuras2
{
   // ejecuta la aplicación
   public static void main( String args[] )
   {
      // crea marco para Figuras2JPanel
      JFrame marco = new JFrame( "Dibujo de figuras en 2D" );
      marco.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );

      Figuras2JPanel figuras2JPanel = new Figuras2JPanel();
      marco.add( figuras2JPanel ); // agrega figuras2JPanel al marco
      marco.setBackground( Color.WHITE ); // establece color de fondo del marco
      marco.setSize( 400, 400 ); // establece el tamaño del marco
      marco.setVisible( true ); // muestra el marco
```

Figura 12.32 | Creación de un objeto `JFrame` para mostrar estrellas. (Parte 1 de 2).

```
    } // fin de main
} // fin de la clase Figuras2
```

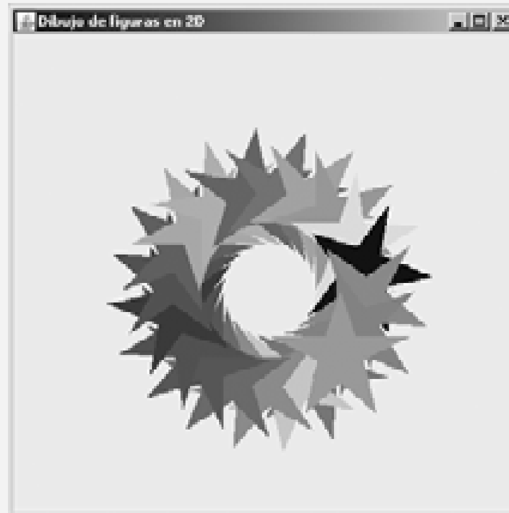


Figura 12.32 | Creación de un objeto `JFrame` para mostrar estrellas. (Parte 2 de 2).

En las líneas 18 y 19 se declaran dos arreglos `int` que representan las coordenadas *x* y *y* de los puntos en la estrella. En la línea 22 se crea el objeto `GeneralPath` llamado `estrella`. En la línea 25 se utiliza el método **`moveTo`** de `GeneralPath` para especificar el primer punto en la `estrella`. La instrucción `for` de las líneas 28 y 29 utiliza el método **`lineTo`** de `GeneralPath` para dibujar una línea al siguiente punto en la `estrella`. Cada nueva llamada a `lineTo` dibuja una línea del punto anterior al punto actual. En la línea 31 se utiliza el método **`closePath`** de `GeneralPath` para dibujar una línea del último punto hasta el punto especificado en la última llamada a `moveTo`. Esto completa la ruta general.

En la línea 33 se utiliza el método **`translate`** de `Graphics2D` para desplazar el origen del dibujo hasta la ubicación (200, 200). Todas las operaciones de dibujo utilizarán ahora la ubicación (200, 200) como si fuera (0, 0).

La instrucción `for` de las líneas 36 a 45 dibujan la estrella 20 veces, girándola alrededor del nuevo punto del origen. En la línea 38 se utiliza el método **`rotate`** de `Graphics2D` para girar la siguiente figura a mostrar. El argumento especifica el ángulo de giro en radianes (con 360° = 2π radianes). En la línea 44 se utiliza el método `fill` de `Graphics2D` para dibujar una versión rellena de la `estrella`.

12.9 Conclusión

En este capítulo aprendió a utilizar las herramientas de gráficos de Java para producir dibujos a colores. Aprendió a especificar la ubicación de un objeto, usando el sistema de coordenadas de Java, y a dibujar en una ventana usando el método `paintComponent`. Vio una introducción a la clase `Color`, y aprendió a utilizar esta clase para especificar distintos colores, usando sus componentes RGB. Utilizó el cuadro de diálogo `JColorChooser` para permitir a los usuarios seleccionar colores en un programa. Después aprendió a trabajar con los tipos de letra al dibujar texto en una ventana. Aprendió a crear un objeto `Font` a partir de un nombre, estilo y tamaño de tipo de letra, así como acceder a la métrica de un tipo de letra. De ahí, aprendió a dibujar varias figuras en una ventana, como rectángulos (regulares, redondeados y en 3D), óvalos y polígonos, así como líneas y arcos. Después utilizó la API Java 2D para crear figuras más complejas y rellenarlas con degradados o patrones. El capítulo concluyó con una discusión sobre las rutas generales, que se utilizan para construir figuras a partir de líneas rectas y curvas complejas. En el siguiente capítulo, aprenderá acerca de las excepciones, que son útiles para manejar errores durante la ejecución de un programa. De esta manera, los errores por manejo proporcionan programas más robustos.

Resumen

Sección 12.1 Introducción

- El sistema de coordenadas de Java es un esquema para identificar todos los posibles puntos en la pantalla.
- Un par de coordenadas está compuesto de una coordenada *x* (la coordenada horizontal) y una coordenada *y* (la coordenada vertical).
- Para mostrar texto y figuras en la pantalla, se especifican sus coordenadas. Estas coordenadas se utilizan para indicar en dónde deben mostrarse los gráficos en una pantalla.
- Las unidades de las coordenadas se miden en píxeles. Un píxel es la unidad más pequeña de resolución de un monitor de computadora.

Sección 12.2 Contextos y objetos de gráficos

- Un contexto de gráficos en Java permite dibujar en la pantalla.
- La clase `Graphics` contiene métodos para dibujar cadenas, líneas, rectángulos y otras figuras. También se incluyen métodos para manipular tipos de letra y colores.
- Un objeto `Graphics` administra un contexto de gráficos y dibuja píxeles en la pantalla, los cuales representan a otros objetos gráficos (por ejemplo, líneas, elipses, rectángulos y otros polígonos.
- La clase `Graphics` es una clase `abstract`. Esto contribuye a la portabilidad de Java; cuando se implementa Java en una plataforma, se crea una subclase de `Graphics`, la cual implementa las herramientas de dibujo. Esta implementación se oculta de nosotros mediante la clase `Graphics`, la cual proporciona la interfaz que nos permite utilizar los gráficos en forma independiente de la plataforma.
- La clase `JComponent` contiene un método `paintComponent` que puede usarse para dibujar los gráficos en un componente de Swing.
- El método `paintComponent` recibe como argumento un objeto `Graphics` que el sistema pasa al método `paintComponent` cuando un componente ligero de Swing necesita volver a pintarse.
- Pocas veces es necesario que el programador llame al método `paintComponent` directamente, ya que el dibujo de gráficos es un proceso controlado por eventos. Cuando se ejecuta una aplicación, el contenedor de ésta llama al método `paintComponent`. Para que `paintComponent` se llame otra vez, debe ocurrir un evento.
- Cuando se muestra un objeto `JComponent`, se hace una llamada a su método `paintComponent`.
- Los programadores llaman al método `repaint` para actualizar los gráficos que se dibujan en el componente de Swing.

Sección 12.3 Control de colores

- La clase `Color` declara métodos y constantes para manipular los colores en un programa.
- Todo color se crea a partir de un componente rojo, uno verde y uno azul. En conjunto estos componentes se llaman valores RGB.
- Los componentes RGB especifican la cantidad de rojo, verde y azul en un color respectivamente. Entre mayor sea el valor RGB, mayor será la cantidad de ese color específico.
- Los métodos `getRed`, `getGreen` y `getBlue` de `Color` devuelven valores enteros de 0 a 255, los cuales representan la cantidad de rojo, verde y azul, respectivamente.
- El método `getColor` de `Graphics` devuelve un objeto `Color` que representa el color actual para dibujar.
- El método `setColor` de graphics establece el color actual para dibujar.
- El método `fillRect` de `Graphics` dibuja un rectángulo relleno por el color actual del objeto `Graphics`.
- El método `drawString` de `Graphics` dibuja un objeto `String` en el color actual.
- El componente de GUI `JColorChooser` permite a los usuarios de una aplicación seleccionar colores.
- La clase `JColorChooser` proporciona el método de conveniencia `static` llamado `showDialog`, que crea un objeto `JColorChooser`, lo adjunta a un cuadro de diálogo y muestra ese cuadro de diálogo.
- Mientras el cuadro de diálogo de selección de color esté en la pantalla, el usuario no podrá interactuar con el componente padre. A este tipo de cuadro de diálogo se le llama cuadro de diálogo modal.

Sección 12.4 Control de tipos de letra

- La clase `Font` contiene métodos y constantes para manipular tipos de letra.
- El constructor de la clase `Font` recibe tres argumentos: el nombre, estilo y tamaño del tipo de letra.

- El estilo de tipo de letra de un objeto Font puede ser Font.PLAIN, Font.ITALIC o Font.BOLD (cada uno es un campo static de la clase Font). Los estilos de tipos de letra pueden usarse combinados (por ejemplo, Font.ITALIC + Font.BOLD).
- El tamaño de un tipo de letra se mide en puntos. Un punto es 1/72 de una pulgada.
- El método setFont de Graphics establece el tipo de letra para dibujar el texto que se va a mostrar.
- El método getStyle de Font devuelve un valor entero que representa el estilo actual del objeto Font.
- El método getSize de Font devuelve el tamaño del tipo de letra, en puntos.
- El método getName de Font devuelve el nombre del tipo de letra actual, como una cadena.
- El método getFamily de Font devuelve el nombre de la familia a la que pertenece el tipo de letra actual. El nombre de la familia del tipo de letra es específico de cada plataforma.
- La clase FontMetrics contiene métodos para obtener información sobre los tipos de letra.
- La métrica de tipos de letra incluye la altura, el descendente (la distancia entre la base de la línea y el punto inferior del tipo de letra), el ascendente (la cantidad que se eleva un carácter por encima de la base de la línea) y el interlineado (la diferencia entre el descendente de una línea de texto y el ascendente de la línea de texto que está arriba; es decir, el espaciamiento entre líneas).

Sección 12.5 Dibujo de líneas, rectángulos y óvalos

- Los métodos fillRoundRect y drawRoundRect de Graphics dibujan rectángulos con esquinas redondeadas.
- Los métodos draw3DRect y fill3DRect de Graphics dibujan rectángulos tridimensionales.
- Los métodos drawOval y fillOval de Graphics dibujan óvalos.

Sección 12.6 Dibujo de arcos

- Un arco se dibuja como una porción de un óvalo.
- Los arcos se extienden desde un ángulo inicial, según el número de grados especificados por el ángulo del arco.
- Los métodos drawArc y fillArc de Graphics se utilizan para dibujar arcos.

Sección 12.7 Dibujo de polígonos y polilíneas

- La clase Polygon contiene métodos para crear polígonos.
- Los polígonos son figuras con varios lados, compuestas de segmentos de línea recta.
- Las polilíneas son una secuencia de puntos conectados.
- El método drawPolyline de Graphics muestra una serie de líneas conectadas.
- Los métodos drawPolygon y fillPolygon de Graphics se utilizan para dibujar polígonos.
- El método addPoint de la clase Polygon agrega pares de coordenadas *x* y *y* a un objeto Polygon.

Sección 12.8 La API Java 2D

- La API Java 2D proporciona herramientas de gráficos bidimensionales avanzadas para los programadores que requieren manipulaciones de gráficos detallados y complejos.
- La clase Graphics2D, que extiende a la clase Graphics, se utiliza para dibujar con la API Java 2D.
- La API Java 2D contiene varias clases para dibujar figuras, incluyendo Line2D.Double, Rectangle2D.Double, Rectangle2D.Double, Arc2D.Double y Ellipse2D.Double.
- La clase GradientPaint ayuda a dibujar una figura en colores que cambian en forma gradual; a esto se le conoce como degradado.
- El método fill de Graphics2D dibuja un objeto Shape relleno; un objeto que implementa a la interfaz Shape.
- La clase BasicStroke ayuda a especificar las características de dibujo de líneas.
- El método draw de Graphics2D se utiliza para dibujar un objeto Shape.
- Las clases GradientPaint y TexturePaint ayudan a especificar las características para rellenar figuras con colores o patrones.
- Una ruta general es una figura construida a partir de líneas rectas y curvas complejas.
- Una ruta general se representa mediante un objeto de la clase GeneralPath.
- El método moveTo de GeneralPath especifica el primer punto en una ruta general.
- El método lineTo de GeneralPath dibuja una línea hasta el siguiente punto en la ruta. Cada nueva llamada a lineTo dibuja una línea desde el punto anterior hasta el punto actual.
- El método closePath de GeneralPath dibuja una línea desde el último punto hasta el punto especificado en la última llamada a moveTo. Esto completa la ruta general.
- El método translate de Graphics2D se utiliza para mover el origen de dibujo hasta una nueva ubicación.
- El método rotate de Graphics2D se utiliza para girar la siguiente figura a mostrar.

Terminología

addPoint, miembro de la clase Polygon
altura (métrica de tipos de letra)
ángulo de un arco
ángulo inicial
arco
ascendente (métrica de tipos de letra)
BasicStroke, clase
BOLD, constante de la clase Font
CAP_ROUND, constante de la clase BasicStroke
clearRect, método de la clase Graphics
closePath, método de la clase GeneralPath
Color, clase
coordenada horizontal
coordenada vertical
coordenada *x*
coordenada *x* superior izquierda
coordenada *y*
coordenada *y* superior izquierda
createGraphics, método de la clase BufferedImage
cuadro de diálogo modal
curva compleja
degradado
degradado acíclico
degradado cíclico
descendente (métrica de tipos de letra)
draw, método de clase Graphics2D
draw3DRect, método de la clase Graphics
drawArc, método de la clase Graphics
drawLine, método de la clase Graphics
drawOval, método de la clase Graphics
drawPolygon, método de la clase Graphics
drawPolyline, método de la clase Graphics
drawRect, método de la clase Graphics
drawRoundRect, método de la clase Graphics
drawString, método de la clase Graphics
eje *x*
eje *y*
esquina superior izquierda de un componente de GUI
Extensión
figuras bidimensionales
fill, método de la clase Graphics2D
fill3DRect, método de la clase Graphics
fillArc, método de la clase Graphics
fillOval, método de la clase Graphics
fillPolygon, método de la clase Graphics
fillRect, método de la clase Graphics
fillRoundRect, método de la clase Graphics
Font, clase
FontMetrics, clase
GeneralPath, clase
getAscent, método de la clase FontMetrics
getBlue, método de la clase Color
getColor, método de la clase Graphics
getDescent, método de la clase FontMetrics
getFamily, método de la clase Font
getFont, método de la clase Graphics
getFontMetrics, método de la clase Graphics
getGreen, método de la clase Color
getHeight, método de la clase FontMetrics
getLeading, método de la clase FontMetrics
getName, método de la clase Font
getRed, método de la clase Color
getSize, método de la clase Font
getStyle, método de la clase Font
GradientPaint, clase
gráficos bidimensionales
Graphics, clase
graphics, contexto
Graphics2D, clase
interlineado (métrica de tipos de letra)
isBold, método de la clase Font
isPlain, método de la clase Font
ITALIC, constante de la clase Font
Java 2D, API
JColorChooser, clase
JOIN_ROUND, constante de la clase BasicStroke
línea base (métrica de tipos de letra)
LinearGradientPaint, clase
líneas conectadas
líneas punteadas
lineTo, método de la clase GeneralPath
manipulación de colores
métrica de tipos de letra
moveTo, método de la clase GeneralPath
muestras de colores
óvalo
óvalo delimitado por un rectángulo
Paint, objeto
paintComponent, método de la clase JComponent
patrón de relleno
píxel ("elemento de imagen")
PLAIN, constante de la clase Font
polígono
polígonos cerrados
polilíneas
Polygon, clase
punto (tamaño de tipo de letra)
RadialGradientPaint, clase
rectángulo con relieve
rectángulos delimitados
rectángulo redondeado
rectángulo tridimensional
Repaint, método de la clase JComponent
RGB valor
Rotate de la clase Graphics2D
ruta general
setBackground, método de la clase Component
setColor, método de la clase Graphics

`setFont`, método de la clase `Graphics`
`setPaint`, método de la clase `Graphics2D`
`setStroke`, método de la clase `Graphics2D`
`showDialog`, método de la clase `JColorChooser`
sistema de coordenadas
`Stroke`, objeto
`TexturePaint`, clase
`translate`, método de la clase `Graphics2D`
unión de líneas

Ejercicios de autoevaluación

12.1 Complete las siguientes oraciones:

a) En Java2D, el método_________________ de la clase _________________ establece las características de una línea utilizada para dibujar una figura.
b) La clase _________________ ayuda a especificar el relleno para una figura, de tal forma que el relleno cambie gradualmente de un color a otro.
c) El método _________________ de la clase `Graphics` dibuja una línea entre dos puntos.
d) RGB son las iniciales de _________________, _________________ y _________________.
e) Los tamaños de los tipos de letra se miden en unidades llamadas _________________.
f) La clase _________________ ayuda a especificar el relleno para una figura, utilizando un patrón dibujado en un objeto `BufferedImage`.

12.2. Conteste con *verdadero* o *falso* a cada una de las siguientes proposiciones; en caso de ser *falso*, explique por qué.

a) Los primeros dos argumentos del método `drawOval` de `Graphics` especifican la coordenada central del óvalo.
b) En el sistema de coordenadas de Java, los valores de *x* se incrementan de izquierda a derecha.
c) El método `fillPolygon` de `Graphics` dibuja un polígono sólido en el color actual.
d) El método `drawArc` de `Graphics` permite ángulos negativos.
e) El método `getSize` de `Graphics` devuelve el tamaño del tipo de letra actual, en centímetros.
f) La coordenada de píxel (0, 0) se encuentra exactamente en el centro del monitor.

12.3 Encuentre el (los) error (es) en cada una de las siguientes instrucciones, y explique cómo corregirlos. Suponga que `g` es un objeto `Graphics`.

a) `g.setFont( "SansSerif" );`
b) `g.erase( x, y, w, h );    // borrar rectángulo en (x, y)`
c) `Font f = new Font( "Serif", Font.BOLDITALIC, 12 );`
d) `g.setColor( 255, 255, 0 );  // cambiar color a amarillo`

Respuestas a los ejercicios de autoevaluación

12.1 a) `setStroke`, `Graphics2D`. b) `GradientPath`. c) `drawLine`. d) rojo, verde, azul. e) puntos. f) `TexturePaint`.

12.2 a) Falso. Los primeros dos argumentos especifican la esquina superior izquierda del rectángulo delimitador.
b) Verdadero.
c) Verdadero.
d) Verdadero.
e) Falso. Los tamaños de los tipos de letra se miden en puntos.
f) Falso. La coordenada (0, 0) corresponde a la esquina superior izquierda de un componente de la GUI, en el cual ocurre el dibujo.

12.3 a) El método `setFont` toma un objeto `Font` como argumento, no un `String`.
b) La clase `Graphics` no tiene un método `erase`. Debe utilizarse el método `clearRect`.
c) `Font.BOLDITALIC` no es un estilo de tipo de letra válido. Para obtener un tipo de letra en cursiva y negrita, use `Font.BOLD + Font.ITALIC`.
d) El método `setColor` toma un objeto `Color` como argumento, no tres enteros.

Ejercicios

12.4 Complete las siguientes oraciones:

a) La clase ________________ de la API Java 2D se utiliza para dibujar óvalos.

b) Los métodos `draw` y `fill` de la clase `Graphics2D` requieren un objeto de tipo ________________ como su argumento.

c) Las tres constantes que especifican el estilo de los tipos de letra son ________________ , ______________ y ________________.

d) El método ________________ de `Graphics2D` establece el color para pintar en figuras de Java2D.

12.5 Conteste con *verdadero* o *falso* a cada una de las siguientes proposiciones; en caso de ser *falso*, explique por qué.

a) El método `drawPolygon` de `Graphics` conecta automáticamente los puntos de los extremos del polígono.

b) El método `drawLine` de `Graphics` dibuja una línea entre dos puntos.

c) El método `fillArc` de `Graphics` utiliza grados para especificar el ángulo.

d) En el sistema de coordenadas de Java, los valores del eje *y* se incrementan de izquierda a derecha.

e) La clase `Graphics` hereda directamente de la clase `Object`.

f) La clase `Graphics` es una clase `abstract`.

g) La clase `Font` hereda directamente de la clase `Graphics`.

12.6 *(Círculos concéntricos mediante el uso del método `drawArc`)* Escriba un programa que dibuje una serie de ocho círculos concéntricos. Los círculos deberán estar separados por 10 píxeles. Use el método `drawOval` de la clase `Graphics`.

12.7 *(Círculos concéntricos mediante el uso de la clase `Ellipse2D.Double`)* Modifique su solución al ejercicio 12.6, para dibujar los óvalos mediante el uso de instancias de la clase `Ellipse2D.Double` y el método `draw` de la clase `Graphics2D`.

12.8 *(Líneas aleatorias mediante el uso de la clase `Line2D.Double`)* Modifique su solución al ejercicio 12.7 para dibujar líneas aleatorias en colores aleatorios y grosores de línea aleatorios. Use la clase `Line2D.Double` y el método `draw` de la clase `Graphics2D` para dibujar las líneas.

12.9 *(Triángulos aleatorios)* Escriba una aplicación que muestre triángulos generados al azar en distintos colores. Cada triángulo deberá rellenarse con un color distinto. Use la clase `GeneralPath` y el método `fill` de la clase `Graphics2D` para dibujar los triángulos.

12.10 *(Caracteres aleatorios)* Escriba un programa que dibuje caracteres al azar, en distintos tamaños y colores de tipo de letra.

12.11 *(Cuadrícula mediante el uso del método `drawLine`)* Escriba una aplicación que dibuje una cuadrícula de 8 por 8. Use el método `drawLine` de `Graphics`.

12.12 *(Cuadrícula mediante el uso de la clase `Line2D.Double`)* Modifique su solución al ejercicio 12.11 para dibujar la cuadrícula utilizando instancias de la clase `Line2D.Double` y el método `draw` de la clase `Graphics2D`.

12.13 *(Cuadrícula mediante el uso del método `drawRect`)* Escriba una aplicación que dibuje una cuadrícula de 10 por 10. Use el método `drawRect` de `Graphics`.

12.14 *(Cuadrícula mediante el uso de la clase* `Rectangle2D.Double`*)* Modifique su solución al ejercicio 12.13 para dibujar la cuadrícula utilizando instancias de la clase `Rectangle2D.Double` y el método `draw` de la clase `Graphics2D`.

12.15 *(Dibujo de tetraedros)* Escriba una aplicación que dibuje un tetraedro (una figura tridimensional con cuatro caras triangulares). Use la clase `GeneralPath` y el método `draw` de la clase `Graphics2D`.

12.16 *(Dibujo de cubos)* Escriba una aplicación que dibuje un cubo. Use la clase `GeneralPath` y el método `draw` de la clase `Graphics2D`.

12.17 *(Círculo mediante el uso de la clase `Ellipse2D.Double`)* Escriba una aplicación que pida al usuario introducir el radio de un círculo como número de punto flotante y que dibuje el círculo, así como los valores del diámetro, la circunferencia y el área del círculo. Use el valor 3.14159 para π. [*Nota:* también puede usar la constante predefinida `Math.PI` para el valor de π. Esta constante es más precisa que el valor 3.14159. La clase `Math` se declara en el paquete `java.lang`, por lo que no necesita importarla]. Use las siguientes fórmulas (*r* es el radio):

$$\textit{diámetro} = 2r$$
$$\textit{circunferencia} = 2\pi r$$
$$\textit{área} = \pi r^2$$

El usuario debe introducir también un conjunto de coordenadas además del radio. Después dibuje el círculo y muestre su diámetro, circunferencia y área, mediante el uso de un objeto `Ellipse2D.Double` para representar el círculo y el método `draw` de la clase `Graphics2D` para mostrarlo en pantalla.

12.18 *(Protector de pantalla)* Escriba una aplicación que simule un protector de pantalla. La aplicación deberá dibujar líneas al azar, utilizando el método `drawLine` de la clase `Graphics`. Después de dibujar 100 líneas, la aplicación deberá borrarse a sí misma y empezar a dibujar líneas nuevamente. Para permitir al programa dibujar en forma continua, coloque una llamada a `repaint` como la última línea en el método `paintComponent`. ¿Observó algún problema con esto en su sistema?

12.19 *(Protector de pantalla mediante el uso de `Timer`)* El paquete `javax.swing` contiene una clase llamada `Timer`, la cual es capaz de llamar al método `actionPerformed` de la interfaz `ActionListener` durante un intervalo de tiempo fijo (especificado en milisegundos). Modifique su solución al ejercicio 12.18 para eliminar la llamada a `repaint` desde el método `paintComponent`. Declare su clase de manera que implemente a `ActionListener`. (El método `actionPerformed` deberá simplemente llamar a `repaint`). Declare una variable de instancia de tipo `Timer`, llamada `temporizador`, en su clase. En el constructor para su clase, escriba las siguientes instrucciones:

```
temporizador = new Timer( 1000, this );
temporizador.start();
```

Esto crea una instancia de la clase `Timer` que llamará al método `actionPerformed` del objeto `this` cada 1000 milisegundos (es decir, cada segundo).

12.20 *(Protector de pantalla para un número aleatorio de líneas)* Modifique su solución al ejercicio 12.19 para permitir al usuario introducir el número de líneas aleatorias que deben dibujarse antes de que la aplicación se borre a sí misma y empiece a dibujar líneas otra vez. Use un objeto `JTextField` para obtener el valor. El usuario deberá ser capaz de escribir un nuevo número en el objeto `JTextField` en cualquier momento durante la ejecución del programa. Use una clase interna para realizar el manejo de eventos para el objeto `JTextField`.

12.21 *(Protector de pantalla con figuras)* Modifique su solución al ejercicio 12.19, de tal forma que utilice la generación de números aleatorios para seleccionar diferentes figuras a mostrar. Use métodos de la clase `Graphics`.

12.22 *(Protector de pantalla mediante el uso de la API Java 2D)* Modifique su solución al ejercicio 12.21 para utilizar clases y herramientas de dibujo de la API Java 2D. Para las figuras como rectángulos y elipses, dibújelas con degradados generados al azar. Use la clase `GradientPaint` para generar el degradado.

12.23 *(Gráficos de tortuga)* Modifique su solución al ejercicio 7.21 (*Gráficos de tortuga*) para agregar una interfaz gráfica de usuario, mediante el uso de objetos `JTextField` y `JButton`. Dibuje líneas en vez de asteriscos (*). Cuando el programa de gráficos de tortuga especifique un movimiento, traduzca el número de posiciones en un número de píxeles en la pantalla, multiplicando el número de posiciones por 10 (o cualquier valor que usted elija). Implemente el dibujo con características de la API Java 2D.

12.24 *(Paseo del caballo)* Produzca una versión gráfica del problema del Paseo del caballo (ejercicios 7.22, 7.23 y 7.26). A medida que se realice cada movimiento, la celda apropiada del tablero de ajedrez deberá actualizarse con el número de movimiento apropiado. Si el resultado del programa es un *paseo completo* o un *paseo cerrado*, el programa deberá mostrar un mensaje apropiado. Si lo desea, puede utilizar la clase `Timer` (vea el ejercicio 12.19) para que le ayude con la animación del Paseo del caballo.

12.25 (*La tortuga y la liebre)* Produzca una versión gráfica de la simulación *La tortuga y la liebre* (ejercicio 7.28). Simule la montaña dibujando un arco que se extienda desde la esquina inferior izquierda de la ventana, hasta la esquina superior derecha. La tortuga y la liebre deberán correr hacia arriba de la montaña. Implemente la salida gráfica de manera que la tortuga y la liebre se impriman en el arco, en cada movimiento. [*Nota:* extienda la longitud de la carrera de 70 a 300, para que cuente con un área de gráficos más grande].

12.26 *(Dibujo de espirales)* Escriba un programa que utilice el método `drawPolyline` de `Graphics` para dibujar una espiral similar a la de la figura 12.33.

12.27 *(Gráfico de pastel)* Escriba un programa que reciba como entrada cuatro números y que los grafique en forma de gráfico de pastel. Use la clase `Arc2D.Double` y el método `fill` de la clase `Graphics2D` para realizar el dibujo. Dibuje cada pieza del pastel en un color distinto.

12.28 *(Selección de figuras)* Escriba una aplicación que permita al usuario seleccionar una figura de un objeto `JComboBox` y que la dibuje 20 veces, con ubicaciones y medidas aleatorias en el método `paintComponent`. El primer elemento en el objeto `JComboBox` debe ser la figura predeterminada a mostrar la primera vez que se hace una llamada a `paintComponent`.



Figura 12.33 | Dibujo de una espiral mediante el uso del método `drawPolyline`.

12.29 *(Colores aleatorios)* Modifique el ejercicio 12.28 para dibujar cada una de las 20 figuras con tamaños aleatorios en un color seleccionado al azar. Use los 13 objetos `Color` predefinidos en un arreglo de objetos `Color`.

12.30 *(Cuadro de diálogo `JColorChooser`)* Modifique el ejercicio 12.28 para permitir al usuario seleccionar de un cuadro de diálogo `JColorChooser` el color en el que deben dibujarse las figuras.

(Opcional) Ejemplo gráfico de GUI y Gráficos: agregar Java 2D

12.31 Java 2D presenta muchas nuevas herramientas para crear gráficos únicos e impresionantes. Agregaremos un pequeño subconjunto de estas características a la aplicación de dibujo que creó en el ejercicio 11.18. En esta versión de la aplicación de dibujo, permitirá al usuario especificar degradados para rellenar figuras y modificar las características de trazo para dibujar líneas y los contornos de las figuras. El usuario podrá elegir cuáles colores van a formar el degradado, y también podrá establecer la anchura y longitud de guión del trazo.

Primero debe actualizar la jerarquía `MiFigura` para que soporte la funcionalidad de Java 2D. Haga las siguientes modificaciones en la clase `MiFigura`:

a) Cambie el tipo del parámetro del método `abstract Dra.`, de `Graphics` a `Graphics2D`.
b) Cambie todas las variables de tipo `Color` al tipo `Saint`, para habilitar el soporte para los degradados. [*Nota:* recuerde que la clase `Color` implementa a la interfaz `Paint`].
c) Agregue una variable de instancia de tipo `Stroke` en la clase `MiFigura` y un parámetro `Stroke` en el constructor, para inicializar la nueva variable de instancia. El trazo predeterminado deberá ser una instancia de la clase `BasicStroke`.

Cada una de las clases `MiLinea`, `MiFiguraDelimitada`, `MiOvalo` y `MiRect` deben agregar un parámetro `Stroke` a sus constructores. En los métodos `draw`, cada figura debe establecer los objetos `Paint` y `Stroke` antes de dibujar o rellenar una figura. Como `Graphics2D` es una subclase de `Graphics`, podemos seguir utilizando los métodos `drawLine`, `drawOval`, `fillOval`, y otros métodos más de `Graphics`, para dibujar las figuras. Al llamar a estos métodos, dibujarán la figura apropiada usando las opciones especificadas para los objetos `Paint` y `Stroke`.

Después, actualice el objeto `PanelDibujo` para manejar las herramientas de Java 2D. Cambie todas las variables `Color` a variables `Saint`. Declare una variable de instancia llamada `trazoActual` de tipo `Stroke`, y proporcione un método *establecer* para esta variable. Actualice las llamadas a los constructores de cada figura para incluir los argumentos `Paint` y `Stroke`. En el método `paintComponent`, convierta la referencia `Graphics` al tipo `Graphics2D` y use la referencia `Graphics2D` en cada llamada al método `draw` de `MiFigura`.

A continuación, haga que se pueda tener acceso a las nuevas características de Java 2D mediante la GUI. Cree un objeto `JPanel` de componentes de GUI para establecer las opciones de Java 2D. Agregue esos componentes a la parte superior del objeto `MarcoDibujo`, debajo del panel que actualmente contiene los controles de las figuras estándar (vea la figura 12.34). Estos componentes de GUI deben incluir lo siguiente:

a) Una casilla de verificación para especificar si se va a pintar usando un degradado.
b) Dos objetos `JButton`, cada uno de los cuales debe mostrar un cuadro de diálogo `JColorChooser`, para permitir al usuario elegir los colores primero y segundo en el degradado. (Éstos sustituirán al objeto `JComboBox` que se utiliza para extender el color en el ejercicio 11.18).
c) Un campo de texto para introducir la anchura del objeto `Stroke`.
d) Un campo de texto para introducir la longitud de guión del objeto `Stroke`.
e) Una casilla de verificación para seleccionar si se va a dibujar una línea punteada o sólida.

Si el usuario opta por dibujar con un degradado, establezca el objeto `Paint` en el `PanelDibujo` de forma que sea un degradado de los dos colores seleccionados por el usuario. La expresión

```
new GradientPaint( 0, 0, color1, 50, 50, color2, true ) )
```

crea un objeto `GradientPath` que avanza diagonalmente en círculos, desde la esquina superior izquierda hasta la esquina inferior derecha, cada 50 píxeles. Las variables `color1` y `color2` representan los colores elegidos por el usuario. Si éste no elije usar un degradado, entonces simplemente establezca el objeto `Paint` en el `PanelDibujo` de manera que sea el primer `Color` elegido por el usuario.

Para los trazos, si el usuario elije una línea sólida, entonces cree el objeto `Stroke` con la expresión

```
new BasicStroke( anchura, BasicStroke.CAP_ROUND, BasicStroke.JOIN_ROUND)
```

en donde la variable `anchura` es la anchura especificada por el usuario en el campo de texto de anchura de línea. Si el usuario selecciona una línea punteada, entonces cree el objeto `Stroke` con la expresión

```
new BasicStroke( anchura, BasicStroke.CAP_ROUND, BasicStroke.JOIN_ROUND, 10, guiones, 0 )
```

en donde `anchura` es de nuevo la anchura en el campo de anchura de texto, y `guiones` es un arreglo con un elemento, cuyo valor es la longitud especificada en el campo de longitud de guión. Los objetos `Panel` y `Stroke` deben pasarse al constructor del objeto figura, cuando se cree la figura en el objeto `PanelDibujo`.

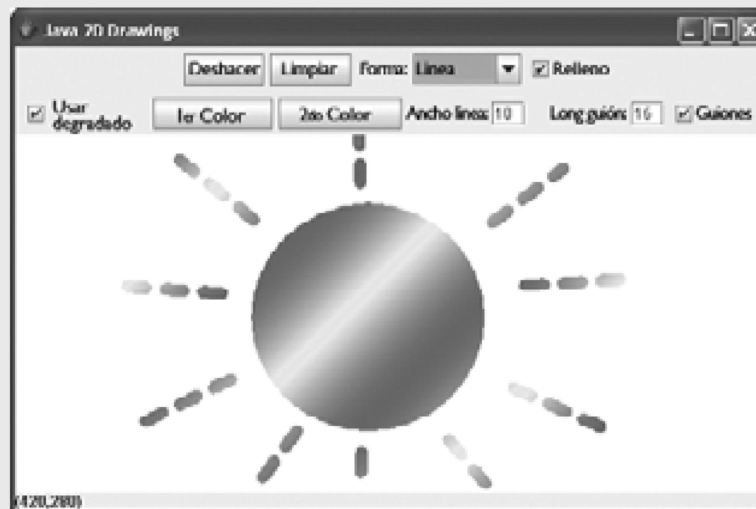
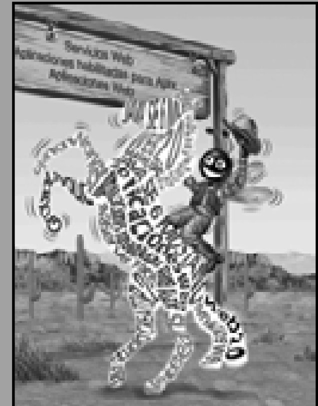


Figura 12.34 | Dibujo con Java 2D.

13

Manejo de excepciones



Es cuestión de sentido común tomar un método y probarlo. Si falla, admítalo francamente y pruebe otro. Pero sobre todo, inténtelo.

—Franklin Delano Roosevelt

¡Oh! Arroja la peor parte de ello, y vive en forma más pura con la otra mitad.

—William Shakespeare

Si están corriendo y no saben hacia dónde se dirigen tengo que salir de alguna parte y atraparlos.

—Jerome David Salinger

¡Oh!.Infinita virtud! ¿Cómo sonríes desde la trampa más grande del mundo sin estar atrapada?

—William Shakespeare

OBJETIVOS

En este capítulo aprenderá a:

- Comprender el manejo de excepciones y de errores.
- Utilizar `try`, `throw` y `catch` para detectar, indicar y manejar excepciones, respectivamente.
- Utilizar el bloque `finally` para liberar recursos.
- Comprender cómo la limpieza de la pila permite que las excepciones que no se atrapan en un alcance, se atrapen en otro.
- Comprender cómo ayuda la pila en la depuración.
- Comprender cómo se ordenan las excepciones en una jerarquía de clases de excepciones.
- Declarar nuevas clases de excepciones.
- Crear excepciones encadenadas que mantengan la información completa del rastreo de la pila.

Plan general

13.1 Introducción

En este capítulo presentaremos el **manejo de excepciones**. Una **excepción** es la indicación de un problema que ocurre durante la ejecución de un programa. El nombre "excepción" implica que el problema ocurre con poca frecuencia; si la "regla" es que una instrucción generalmente se ejecuta en forma correcta, entonces la "excepción a la regla" es cuando ocurre un problema. El manejo de excepciones le permite crear aplicaciones que puedan resolver (o manejar) las excepciones. En muchos casos, el manejo de una excepción permite que el programa continúe su ejecución como si no se hubiera encontrado el problema. Un problema más grave podría evitar que un programa continuara su ejecución normal, en vez de requerir al programa que notifique al usuario sobre el problema antes de terminar de una manera controlada. Las características que presentamos en este capítulo permiten a los programadores escribir **programas tolerantes a fallas** y **robustos** (es decir, programas que traten con los problemas que puedan surgir sin dejar de ejecutarse). El estilo y los detalles sobre el manejo de excepciones en Java se basan, en parte, en el trabajo que Andrew Koenig y Bjarne Stroustrup presentaron en su artículo "Exception Handling for C++ (versión revisada)."[1]



Tip para prevenir errores 13.1

El manejo de excepciones ayuda a mejorar la tolerancia a fallas de un programa.

Ya vimos en capítulos anteriores una breve introducción a las excepciones. En el capítulo 7 aprendió que una excepción `ArrayIndexOutOfBoundsException` ocurre cuando hay un intento por acceder a un elemento más allá del fin del arreglo. Dicho problema puede ocurrir si hay un error de "desplazamiento por 1" en una instrucción `for` que manipula un arreglo. En el capítulo 10 presentamos la excepción `ClassCastException`, que ocurre cuando hay un intento por convertir un objeto que no tiene una relación *"es un"* con el tipo especificado en el operador de conversión. En el capítulo 11 hicimos una breve mención de la excepción `NullPointerException`, la cual ocurre cada vez que se utiliza una referencia `null` en donde se espera un objeto (por ejemplo, cuando hay un intento por adjuntar un componente de GUI a un objeto `Container`, pero el componente de GUI no se ha creado todavía). A lo largo de este libro ha utilizado también la clase `Scanner`; que, como veremos en este capítulo, también puede producir excepciones.

El capítulo empieza con un panorama general de los conceptos relacionados con el manejo de excepciones, y posteriormente se demuestran las técnicas básicas para el manejo de excepciones. Mostraremos estas técnicas

---

1. Koenig, A. y B. Stroustrup. "Exception Handling for C++ (versión revisada)", *Proceedings of the Usenix C++ Conference,* pp. 149-176, San Francisco, abril de 1990.

en acción, mediante un ejemplo que señala cómo manejar una excepción que ocurre cuando un método intenta realizar una división entre cero. Después del ejemplo, presentaremos varias clases de la parte superior de la jerarquía de clases de Java para el manejo de excepciones. Como verá posteriormente, sólo las clases que extienden a `Throwable` (paquete `java.lang`) en forma directa o indirecta pueden usarse para manejar excepciones. Después hablaremos sobre la característica de las excepciones encadenadas, que permiten a los programadores envolver la información acerca de una excepción que haya ocurrido en otro objeto de excepción, para proporcionar información más detallada acerca de un problema en un programa. Luego hablaremos sobre ciertas cuestiones adicionales sobre el manejo de excepciones, como la forma en que se deben manejar las excepciones que ocurren en un constructor. Presentaremos las precondiciones y poscondiciones, que deben ser verdaderas cuando se hacen llamadas a sus métodos y cuando esos métodos regresan, respectivamente. Por último presentaremos las aserciones, que los programadores utilizan en tiempo de desarrollo para facilitar el proceso de depurar su código.

13.2 Generalidades acerca del manejo de excepciones

Con frecuencia, los programas evalúan condiciones que determinan cómo debe proceder la ejecución. Considere el siguiente seudocódigo:

Realizar una tarea

Si la tarea anterior no se ejecutó correctamente
  Realizar el procesamiento de los errores

Realizar la siguiente tarea

Si la tarea anterior no se ejecutó correctamente
  Realizar el procesamiento de los errores

…

En este seudocódigo empezamos realizando una tarea; después, evaluamos si esa tarea se ejecutó correctamente. Si no lo hizo, realizamos el procesamiento de los errores. De otra manera, continuamos con la siguiente tarea. Aunque esta forma de manejo de errores funciona, al entremezclar la lógica del programa con la lógica del manejo de errores el programa podría ser difícil de leer, modificar, mantener y depurar; especialmente en aplicaciones extensas.



Tip de rendimiento 13.1

Si los problemas potenciales ocurren con poca frecuencia, al entremezclar la lógica del programa y la lógica del manejo de errores se puede degradar el rendimiento del programa, ya que éste debe realizar pruebas (tal vez con frecuencia) para determinar si la tarea se ejecutó en forma correcta, y si se puede llevar a cabo la siguiente tarea.

El manejo de excepciones permite al programador remover el código para manejo de errores de la "línea principal" de ejecución del programa, lo cual mejora la claridad y capacidad de modificación del mismo. Usted puede optar por manejar las excepciones que elija: todas las excepciones, todas las de cierto tipo o todas las de un grupo de tipos relacionados (por ejemplo, los tipos de excepciones que están relacionados a través de una jerarquía de herencia). Esta flexibilidad reduce la probabilidad de que los errores se pasen por alto y, por consecuencia, hace que los programas sean más robustos.

Con lenguajes de programación que no soportan el manejo de excepciones, los programadores a menudo retrasan la escritura de código de procesamiento de errores, o algunas veces olvidan incluirlo. Esto hace que los productos de software sean menos robustos. Java permite al programador tratar con el manejo de excepciones fácilmente, desde el comienzo de un proyecto.

13.3 Ejemplo: división entre cero sin manejo de excepciones

Demostraremos primero qué ocurre cuando surgen errores en una aplicación que no utiliza el manejo de errores. En la figura 13.1 se pide al usuario que introduzca dos enteros y éstos se pasan al método `cociente`, que calcula el cociente y devuelve un resultado `int`. En este ejemplo veremos que las excepciones se **lanzan** (es decir, la excepción ocurre) cuando un método detecta un problema y no puede manejarlo.

La primera de las tres ejecuciones de ejemplo en la figura 13.1 muestra una división exitosa. En la segunda ejecución de ejemplo, el usuario introduce el valor `0` como denominador. Observe que muestran varias líneas de

información en respuesta a esta entrada inválida. Esta información se conoce como el **rastreo de la pila**, la cual incluye el nombre de la excepción (`java.lang.ArithmeticException`) en un mensaje descriptivo, que indica el problema que ocurrió y la pila de llamadas a métodos completa (es decir, la cadena de llamadas) al momento en que ocurrió la excepción. El rastreo de la pila incluye la ruta de ejecución que condujo a la excepción, método por método. Esta información ayuda a depurar un programa. La primera línea especifica que ha ocurrido una

```
// Fig. 13.1: DivisionEntreCeroSinManejoDeExcepciones.java
// Una aplicación que trata de realizar una división entre cero.
import java.util.Scanner;

public class DivisionEntreCeroSinManejoDeExcepciones
{
   // demuestra el lanzamiento de una excepción cuando ocurre una división entre cero
   public static int cociente( int numerador, int denominador )
   {
      return numerador / denominador; // posible división entre cero
   } // fin del método cociente

   public static void main( String args[] )
   {
      Scanner explorador = new Scanner( System.in ); // objeto Scanner para entrada

      System.out.print( "Introduzca un numerador entero: " );
      int numerador = explorador.nextInt();
      System.out.print( "Introduzca un denominador entero: " );
      int denominador = explorador.nextInt();

      int resultado = cociente( numerador, denominador );
      System.out.printf(
         "\nResultado: %d / %d = %d\n", numerador, denominador, resultado );
   } // fin de main
} // fin de la clase DivisionEntreCeroSinManejoDeExcepciones
```

```
Introduzca un numerador entero: 100
Introduzca un denominador entero: 7

Resultado: 100 / 7 = 14
```

```
Introduzca un numerador entero: 100
Introduzca un denominador entero: 0
Exception in thread “main” java.lang.ArithmeticException: / by zero
        at DivisionEntreCeroSinManejoDeExcepciones.cociente(
           DivisionEntreCeroSinManejoDeExcepciones.java:10)
        at DivisionEntreCeroSinManejoDeExcepciones.main(
           DivisionEntreCeroSinManejoDeExcepciones.java:22)
```

```
Introduzca un numerador entero: 100
Introduzca un denominador entero: hola
Exception in thread “main” java.util.InputMismatchException
        at java.util.Scanner.throwFor(Scanner.java:840)
        at java.util.Scanner.next(Scanner.java:1461)
        at java.util.Scanner.nextInt(Scanner.java:2091)
        at java.util.Scanner.nextInt(Scanner.java:2050)
        at DivisionEntreCeroSinManejoDeExcepciones.main(
           DivisionEntreCeroSinManejoDeExcepciones.java:20)
```

Figura 13.1 | División entera sin manejo de excepciones.

excepción `ArithmeticException`. El texto después del nombre de la excepción ("`/ by zero`") indica que esta excepción ocurrió como resultado de un intento de dividir entre cero. Java no permite la división entre cero en la aritmética de enteros. [*Nota:* Java *sí* permite la división entre cero con valores de punto flotante. Dicho cálculo produce como resultado el valor de infinito, que se representa en Java como un valor de punto flotante (pero en realidad aparece como la cadena `Infinity`)]. Cuando ocurre una división entre cero en la aritmética de enteros, Java lanza una excepción **`ArithmeticException`**. Este tipo de excepciones pueden surgir debido a varios problemas distintos en aritmética, por lo que los datos adicionales ("`/ by zero`") nos proporcionan más información acerca de esta excepción específica.

Empezando a partir de la última línea del rastreo de la pila, podemos ver que la excepción se detectó en la línea 22 del método `main`. Cada línea del rastreo de la pila contiene el nombre de la clase y el método (`DivideByZeroNoExceptionHandling.main`) seguido por el nombre del archivo y el número de línea (`DivideByZeroNoExceptionHandling.java:22`). Siguiendo el rastreo de la pila, podemos ver que la excepción ocurre en la línea 10, en el método `cociente`. La fila superior de la cadena de llamadas indica el **punto de lanzamiento**: el punto inicial en el que ocurre la excepción. El punto de lanzamiento de esta excepción está en la línea 10 del método `cociente`.

En la tercera ejecución, el usuario introduce la cadena `"hola"` como denominador. Observe de nuevo que se muestra un rastreo de la pila. Esto nos informa que ha ocurrido una excepción `InputMismatchException` (paquete `java.util`). En nuestros ejemplos anteriores, en donde se leían valores numéricos del usuario, se suponía que éste debía introducir un valor entero apropiado. Sin embargo, algunas veces los usuarios cometen errores e introducen valores no enteros. Una excepción **`InputMismatchException`** ocurre cuando el método `nextInt` de `Scanner` recibe una cadena que no representa un entero válido. Empezando desde el final del rastreo de la pila, podemos ver que la excepción se detectó en la línea 20 del método `main`. Siguiendo el rastreo de la pila, podemos ver que la excepción ocurre en el método `nextInt`. Observe que en vez del nombre de archivo y del número de línea, se proporciona el texto `Unknown Source`. Esto significa que la JVM no tiene acceso al código fuente en donde ocurrió la excepción.

Observe que en las ejecuciones de ejemplo de la figura 13.1, cuando ocurren excepciones y se muestran los rastreos de la pila, el programa también termina. Esto no siempre ocurre en Java; algunas veces un programa puede continuar, aun cuando haya ocurrido una excepción y se haya impreso un rastreo de pila. En tales casos, la aplicación puede producir resultados inesperados. En la siguiente sección le mostraremos cómo manejar esas excepciones y mantener el programa ejecutándose sin problema.

En la figura 13.1, ambos tipos de excepciones se detectaron en el método `main`. En el siguiente ejemplo, veremos cómo manejar estas excepciones para permitir que el programa se ejecute hasta terminar de manera normal.

13.4 Ejemplo: manejo de excepciones tipo `ArithmeticException` e `InputMismatchException`

La aplicación de la figura 13.2, que se basa en la figura 13.1, utiliza el manejo de excepciones para procesar cualquier excepción tipo `ArithmeticException` e `InputMismatchException` que pueda ocurrir. La aplicación todavía pide dos enteros al usuario y los pasa al método `cociente`, que calcula el cociente y devuelve un resultado `int`. Esta versión de la aplicación utiliza el manejo de excepciones de manera que, si el usuario comete un error, el programa **atrapa** y **maneja** (es decir, se encarga de) la excepción; en este caso, le permite al usuario tratar de introducir los datos de entrada otra vez.

```
// Fig. 13.2: DivisionEntreCeroConManejoDeExcepciones.java
// Un ejemplo de manejo de excepciones que verifica la división entre cero.
import java.util.InputMismatchException;
import java.util.Scanner;

public class DivisionEntreCeroConManejoDeExcepciones
{
   // demuestra cómo se lanza una excepción cuando ocurre una división entre cero
```

Figura 13.2 | Manejo de excepciones `ArithmeticException` e `InputMismatchException`. (Parte 1 de 3).

```
   public static int cociente( int numerador, int denominador )
      throws ArithmeticException
   {
      return numerador / denominador; // posible división entre cero
   } // fin del método cociente

   public static void main( String args[] )
   {
      Scanner explorador = new Scanner( System.in ); // objeto Scanner para entrada
      boolean continuarCiclo = true; // determina si se necesitan más datos de entrada

      do
      {
         try // lee dos números y calcula el cociente
         {
            System.out.print( "Introduzca un numerador entero: " );
            int numerador = explorador.nextInt();
            System.out.print( "Introduzca un denominador entero: " );
            int denominador = explorador.nextInt();

            int resultado = cociente( numerador, denominador );
            System.out.printf( "\nResultado: %d / %d = %d\n", numerador,
               denominador, resultado );
            continuarCiclo = false; // entrada exitosa; termina el ciclo
         } // fin de bloque try
         catch ( InputMismatchException inputMismatchException )
         {
            System.err.printf( "\nExcepcion: %s\n",
               inputMismatchException );
            explorador.nextLine(); // descarta entrada para que el usuario intente otra vez
            System.out.println(
               "Debe introducir enteros. Intente de nuevo.\n" );
         } // fin de bloque catch
         catch ( ArithmeticException arithmeticException )
         {
            System.err.printf( "\nExcepcion: %s\n", arithmeticException );
            System.out.println(
               "Cero es un denominador invalido. Intente de nuevo.\n" );
         } // fin de catch
      } while ( continuarCiclo ); // fin de do...while
   } // fin de main
} // fin de la clase DivisionEntreCeroConManejoDeExcepciones
```

```
Introduzca un numerador entero: 100
Introduzca un denominador entero: 7

Resultado: 100 / 7 = 14
```

```
Introduzca un numerador entero: 100
Introduzca un denominador entero: 0

Excepcion: java.lang.ArithmeticException: / by zero
Cero es un denominador invalido. Intente de nuevo.

Introduzca un numerador entero: 100
Introduzca un denominador entero: 7

Resultado: 100 / 7 = 14
```

Figura 13.2 | Manejo de excepciones `ArithmeticException` e `InputMismatchException`. (Parte 2 de 3).

```
Introduzca un numerador entero: 100
Introduzca un denominador entero: hola

Excepcion: java.util.InputMismatchException
Debe introducir enteros. Intente de nuevo.

Introduzca un numerador entero: 100
Introduzca un denominador entero: 7

Resultado: 100 / 7 = 14
```

Figura 13.2 | Manejo de excepciones `ArithmeticException` e `InputMismatchException`. (Parte 3 de 3).

La primera ejecución de ejemplo de la figura 13.2 muestra una ejecución exitosa que no se encuentra con ningún problema. En la segunda ejecución, el usuario introduce un denominador cero y ocurre una excepción `ArithmeticException`. En la tercera ejecución, el usuario introduce la cadena `"hola"` como el denominador, y ocurre una excepción `InputMismatchException`. Para cada excepción, se informa al usuario sobre el error y se le pide que intente de nuevo; después el programa le pide dos nuevos enteros. En cada ejecución de ejemplo, el programa se ejecuta hasta terminar sin problemas.

La clase `InputMismatchException` se importa en la línea 3. La clase `ArithmeticException` no necesita importarse, ya que se encuentra en el paquete `java.lang`. El método `main` (líneas 15 a 49) crea un objeto `Scanner` en la línea 17. En la línea 18 se crea la variable `bolean` llamada `continuarCiclo`, la cual es verdadera si el usuario no ha introducido aún datos de entrada válidos. En las líneas 20 a 48 se pide repetidas veces a los usuarios que introduzcan datos, hasta recibir una entrada válida.

Encerrar código en un bloque `try`

Las líneas 22 a 33 contienen un **bloque `try`**, que encierra el código que podría lanzar (`throw`) una excepción y el código que no debería ejecutarse en caso de que ocurra una excepción (es decir, si ocurre una excepción, se omitirá el resto del código en el bloque `try`). Un bloque `try` consiste en la palabra clave `try` seguida de un bloque de código, encerrado entre llaves (`{}`). [*Nota:* el término "bloque `try`" se refiere algunas veces sólo al bloque de código que va después de la palabra clave `try` (sin incluir a la palabra `try`). Para simplificar, usaremos el término "bloque `try`" para referirnos al bloque de código que va después de la palabra clave `try`, incluyendo esta palabra]. Las instrucciones que leen los enteros del teclado (líneas 25 y 27) utilizan el método `nextInt` para leer un valor `int`. El método `nextInt` lanza una excepción `InputMismatchException` si el valor leído no es un entero válido.

La división que puede provocar una excepción `ArithmeticException` no se ejecuta en el bloque `try`. En vez de ello, la llamada al método `cociente` (línea 29) invoca al código que intenta realizar la división (línea 12); la JVM lanza un objeto `ArithmeticException` cuando el denominador es cero.



Observación de ingeniería de software 13.1

Las excepciones pueden surgir a través de código mencionado en forma explícita en un bloque `try`, a través de llamadas a otros métodos, de llamadas a métodos con muchos niveles de anidamiento, iniciadas por código en un bloque `try` o desde la Máquina Virtual de Java, al momento en que ejecute códigos de byte de Java.

Atrapar excepciones

El bloque `try` en este ejemplo va seguido de dos bloques `catch`: uno que maneja una excepción `InputMismatchException` (líneas 34 a 41) y uno que maneja una excepción `ArithmeticException` (líneas 42 a 47). Un **bloque `catch`** (también conocido como **cláusula `catch`** o **manejador de excepciones**) atrapa (es decir, recibe) y maneja una excepción. Un bloque `catch` empieza con la palabra clave `catch` y va seguido por un parámetro entre paréntesis (conocido como el parámetro de excepción, que veremos en breve) y un bloque de código encerrado entre llaves. [*Nota:* el término "cláusula `catch`" se utiliza algunas veces para hacer referencia a la palabra clave `catch`, seguida de un bloque de código, en donde el término "bloque `catch`" se refiere sólo al bloque de código que va después de la palabra clave `catch`, sin incluirla. Para simplificar, usaremos el término "bloque `catch`" para referirnos al bloque de código que va después de la palabra clave `catch`, incluyendo esta palabra].

Por lo menos un bloque `catch` o un **bloque `finally`** (que veremos en la sección 13.7) debe ir inmediatamente después del bloque `try`. Cada bloque `catch` especifica entre paréntesis un **parámetro de excepción**, que identifica el tipo de excepción que puede procesar el manejador. Cuando ocurre una excepción en un bloque `try`, el bloque `catch` que se ejecuta es aquél cuyo tipo coincide con el tipo de la excepción que ocurrió (es decir, el tipo en el bloque `catch` coincide exactamente con el tipo de la excepción que se lanzó, o es una superclase de ésta). El nombre del parámetro de excepción permite al bloque `catch` interactuar con un objeto de excepción atrapada; por ejemplo, para invocar en forma implícita el método `toString` de la excepción que se atrapó (como en las líneas 37 y 44), que muestra información básica acerca de la excepción. La línea 38 del primer bloque `catch` llama al método `nextLine` de `Scanner`. Como ocurrió una excepción `InputMismatchException`, la llamada al método `nextInt` nunca leyó con éxito los datos del usuario; por lo tanto, leemos esa entrada con una llamada al método `nextLine`. No hacemos nada con la entrada en este punto, ya que sabemos que es inválida. Cada bloque `catch` muestra un mensaje de error y pide al usuario que intente de nuevo. Al terminar alguno de los bloques `catch`, se pide al usuario que introduzca datos. Pronto veremos con más detalle la manera en que trabaja este flujo de control en el manejo de excepciones.



Error común de programación 13.1

Es un error de sintaxis colocar código entre un bloque `try` y su correspondiente bloque `catch`.



Error común de programación 13.2

Cada instrucción `catch` sólo puede tener un parámetro; especificar una lista de parámetros de excepción separada por comas es un error de sintaxis.

Una **excepción no atrapada** ocurre y no hay bloques `catch` que coincidan. En el segundo y tercer resultado de ejemplo de la figura 13.1, vio las excepciones no atrapadas. Recuerde que cuando ocurrieron excepciones en ese ejemplo, la aplicación terminó antes de tiempo (después de mostrar el rastreo de pila de la excepción). Esto no siempre ocurre como resultado de las excepciones no atrapadas. Como aprenderá en el capítulo 23, Subprocesamiento múltiple, Java utiliza un modelo de ejecución de programas con subprocesamiento múltiple. Cada **subproceso** es una actividad paralela. Un programa puede tener muchos subprocesos. Si un programa sólo tiene un subproceso, una excepción no atrapada hará que el programa termine. Si un programa tiene varios subprocesos, una excepción no atrapada terminará sólo el subproceso en el cual ocurrió la excepción. Sin embargo, en dichos programas ciertos subprocesos pueden depender de otros, y si un subproceso termina debido a una excepción no atrapada, puede haber efectos adversos para el resto del programa.

Modelo de terminación del manejo de excepciones

Si ocurre una excepción en un bloque `try` (por ejemplo, si se lanza una excepción `InputMismatchException` como resultado del código de la línea 25 en la figura 13.2), el bloque `try` termina de inmediato y el control del programa se transfiere al primero de los siguientes bloques `catch` en los que el tipo del parámetro de excepción coincide con el tipo de la excepción que se lanzó. En la figura 13.2, el primer bloque `catch` atrapa excepciones `InputMismatchException` (que ocurren si se introducen datos de entrada inválidos) y el segundo bloque `catch` atrapa excepciones `ArithmeticException` (que ocurren si hay un intento por dividir entre cero). Una vez que se maneja la excepción, el control del programa no regresa al punto de lanzamiento, ya que el bloque `try` ha expirado (y se han perdido sus variables locales). En vez de ello, el control se reanuda después del último bloque `catch`. Esto se conoce como el **modelo de terminación del manejo de excepciones**. [*Nota:* algunos lenguajes utilizan el **modelo de reanudación del manejo de excepciones** en el que, después de manejar una excepción, el control se reanuda justo después del punto de lanzamiento].



Error común de programación 13.3

Pueden ocurrir errores lógicos si usted supone que después de manejar una excepción, el control regresará a la primera instrucción después del punto de lanzamiento.



Tip para prevenir errores 13.2

Con el manejo de excepciones, un programa puede seguir ejecutándose (en vez de terminar) después de lidiar con un problema. Esto ayuda a asegurar el tipo de aplicaciones robustas que contribuyen a lo que se conoce como computación de misión crítica, o computación crítica para los negocios.

Observe que nombramos a nuestros parámetros de excepción (`inputMismatchException` y `arithmeticException`) en base a su tipo. A menudo, los programadores de Java utilizan simplemente la letra `e` como el nombre de sus parámetros de excepción.



Buena práctica de programación 13.1

El uso del nombre de un parámetro de excepción que refleje el tipo del parámetro fomenta la claridad, al recordar al programador el tipo de excepción que se está manejando.

Después de ejecutar un bloque `catch`, el flujo de control de este programa procede a la primera instrucción después del último bloque `catch` (línea 48 en este caso). La condición en la instrucción `do...while` es `true` (la variable `continuarCiclo` contiene su valor inicial de `true`), por lo que el control regresa al principio del ciclo y se le pide al usuario una vez más que introduzca datos. Esta instrucción de control iterará hasta que se introduzcan datos de entrada válidos. En ese punto, el control del programa llega a la línea 32, en donde se asigna `false` a la variable `continuarCiclo`. Después, el bloque `try` termina. Si no se lanzan excepciones en el bloque `try`, se omiten los bloques `catch` y el control continúa con la primera instrucción después de los bloques `catch` (en la sección 13.7 aprenderemos acerca de otra posibilidad, cuando hablemos sobre el bloque `finally`). Ahora la condición del ciclo `do...while` es `false`, y el método `main` termina.

El bloque `try` y sus correspondientes bloques `catch` y/o `finally` forman en conjunto una **instrucción `try`**. Es importante no confundir los términos "bloque `try`" e "instrucción `try`"; el término "bloque `try`" se refiere a la palabra clave `try` seguida de un bloque de código, mientras que "instrucción `try`" incluye el bloque `try`, así como los siguientes bloques `catch` y/o un bloque `finally`.

Al igual que con cualquier otro bloque de código, cuando termina un bloque `try`, se destruyen las variables locales declaradas en ese bloque. Cuando termina un bloque `catch`, las variables locales declaradas dentro de este bloque (incluyendo el parámetro de excepción de ese bloque `catch`) también quedan fuera de alcance y se destruyen. Cualquier bloque `catch` restante en la instrucción `try` se ignora, y la ejecución se reanuda en la primera línea de código después de la secuencia `try...catch`; ésta será un bloque `finally`, en caso de que haya uno presente.

Uso de la cláusula `throws`

Ahora examinaremos el método `cociente` (figura 13.2; líneas 9 a 13). La porción de la declaración del método ubicada en la línea 10 se conoce como **cláusula `throws`**. Esta cláusula especifica las excepciones que lanza el método. La cláusula aparece después de la lista de parámetros del método y antes de su cuerpo. Contiene una lista separada por comas de las excepciones que lanzará el método, en caso de que ocurra un problema. Dichas excepciones pueden lanzarse mediante instrucciones en el cuerpo del método, o mediante métodos que se llamen desde el cuerpo. Un método puede lanzar excepciones de las clases que se listen en su cláusula `throws`, o en la de sus subclases. Hemos agregado la cláusula `throws` a esta aplicación, para indicar al resto del programa que este método puede lanzar una excepción `ArithmeticException`. Por ende, a los clientes del método `cociente` se les informa que el método puede lanzar una excepción `ArithmeticException`. En la sección 13.6 aprenderá más acerca de la cláusula `throws`.



Tip para prevenir errores 13.3

Si sabe que un método podría lanzar una excepción, incluya el código apropiado para manejar excepciones en su programa, para que sea más robusto.



Tip para prevenir errores 13.4

Lea la documentación de la API en línea para saber acerca de un método, antes de usarlo en un programa. La documentación especifica la excepción lanzada por el método (si la hay), y también indica las razones por las que pueden ocurrir dichas excepciones. Después, incluya el código adecuado para manejar esas excepciones en su programa.



Tip para prevenir errores 13.5

Lea la documentación de la API en línea para buscar una clase de excepción, antes de escribir código para manejar ese tipo de excepciones. Por lo general, la documentación para una clase de excepción contiene las razones potenciales por las que podrían ocurrir dichas excepciones durante la ejecución de un programa.

Cuando se ejecuta la línea 12, si el `denominador` es cero, la JVM lanza un objeto `ArithmeticException`. Este objeto será atrapado por el bloque `catch` en las líneas 42 a 47, que muestra información básica acerca de la excepción, invocando de manera implícita al método `toString` de la excepción, y después pide al usuario que intente de nuevo.

Si el `denominador` no es cero, el método `cociente` realiza la división y devuelve el resultado al punto de la invocación, al método `cociente` en el bloque `try` (línea 29). Las líneas 30 y 31 muestran el resultado del cálculo y la línea 32 establece `continuarCiclo` en `false`. En este caso, el bloque `try` se completa con éxito, por lo que el programa omite los bloques `catch` y la condición falla en la línea 48, y el método `main` termina de ejecutarse en forma normal.

Observe que cuando `cociente` lanza una excepción `ArithmeticException`, `cociente` termina y no devuelve un valor, y las variables locales de `cociente` quedan fuera de alcance (y se destruyen). Si `cociente` contiene variables locales que sean referencias a objetos y no hay otras referencias a esos objetos, éstos se marcan para la recolección de basura. Además, cuando ocurre una excepción, el bloque `try` desde el cual se llamó `cociente` termina antes de que puedan ejecutarse las líneas 30 a 32. Aquí también, si las variables locales se crearon en el bloque `try` antes de que se lanzara la excepción, estas variables quedarían fuera de alcance.

Si se genera una excepción `InputMismatchException` mediante las líneas 25 o 27, el bloque `try` termina y la ejecución continúa con el bloque `catch` en las líneas 34 a 41. En este caso, no se hace una llamada al método `cociente`. Entonces, el método `main` continúa después del último bloque `catch` (línea 48).

13.5 Cuándo utilizar el manejo de excepciones

El manejo de excepciones está diseñado para procesar **errores sincrónicos**, que ocurren cuando se ejecuta una instrucción. Ejemplos comunes de estos errores que veremos en este libro son los índices fuera de rango, el desbordamiento aritmético (es decir, un valor fuera del rango representable de valores), la división entre cero, los parámetros inválidos de método, la interrupción de subprocesos y la asignación fallida de memoria (debido a la falta de ésta). El manejo de excepciones no está diseñado para procesar los problemas asociados con los **eventos asíncronos** (por ejemplo, completar las operaciones de E/S de disco, la llegada de mensajes de red, clics del ratón y pulsaciones de teclas), los cuales ocurren en paralelo con, y en forma independiente de, el flujo de control del programa.



Observación de ingeniería de software 13.2

Incorpore su estrategia de manejo de excepciones en sus sistemas, partiendo desde el principio del proceso de diseño. Puede ser difícil incluir un manejo efectivo de las excepciones, después de haber implementado un sistema.



Observación de ingeniería de software 13.3

El manejo de excepciones proporciona una sola técnica uniforme para procesar los problemas. Esto ayuda a los programadores que trabajan en proyectos extensos a comprender el código de procesamiento de errores de los demás programadores.



Observación de ingeniería de software 13.4

Evite usar el manejo de excepciones como una forma alternativa de flujo de control. Estas excepciones "adicionales" pueden "estorbar" a las excepciones de tipos de errores genuinos.



Observación de ingeniería de software 13.5

El manejo de excepciones simplifica la combinación de componentes de software, y les permite trabajar en conjunto con efectividad, al permitir que los componentes predefinidos comuniquen los problemas a los componentes específicos de la aplicación, quienes a su vez pueden procesar los problemas en forma específica para la aplicación.

13.6 Jerarquía de excepciones en Java

Todas las clases de excepciones heredan, ya sea en forma directa o indirecta, de la clase **`Exception`**, formando una jerarquía de herencias. Los programadores pueden extender esta jerarquía para crear sus propias clases de excepciones.

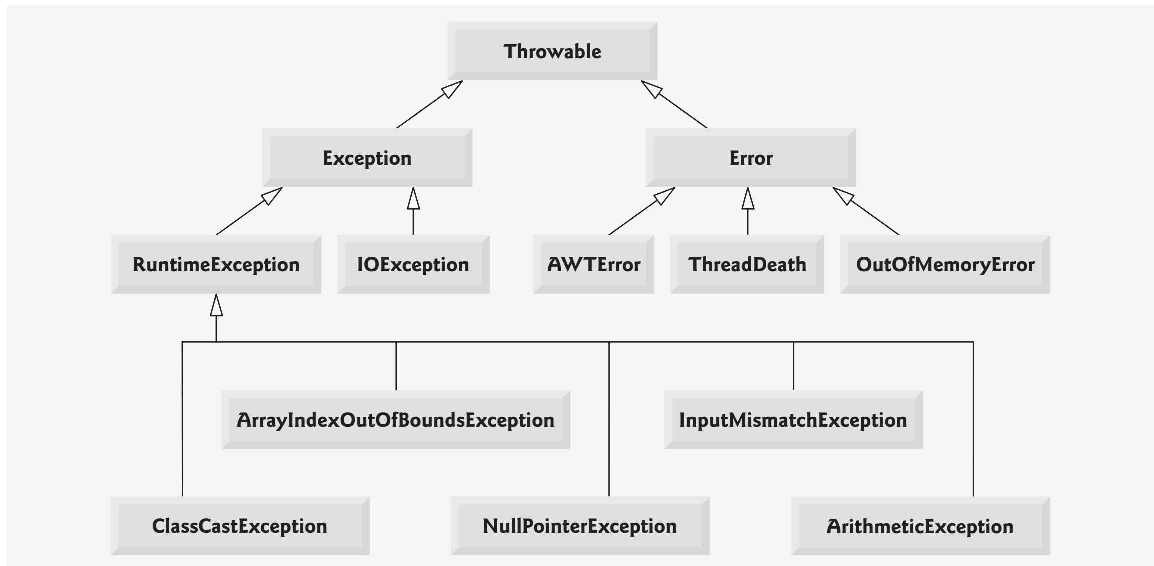


Figura 13.3 | Porción de la jerarquía de herencia de la clase `Throwable`.

La figura 13.3 muestra una pequeña porción de la jerarquía de herencia para la clase **`Throwable`** (una subclase de `Object`), que es la superclase de la clase `Exception`. Sólo pueden usarse objetos `Throwable` con el mecanismo para manejar excepciones. La clase `Throwable` tiene dos subclases: `Exception` y `Error`. La clase `Exception` y sus subclases (por ejemplo, `RuntimeException`, del paquete `java.lang`, e `IOException`, del paquete `java.io`) representan situaciones excepcionales que pueden ocurrir en un programa en Java, y que pueden ser atrapadas por la aplicación. La clase **`Error`** y sus subclases (por ejemplo, `OutOfMemoryError`) representan situaciones anormales que podrían ocurrir en la JVM. Los errores tipo `Error` ocurren con poca frecuencia y no deben ser atrapados por las aplicaciones; por lo general, no es posible que las aplicaciones se recuperen de los errores tipo `Error`. [*Nota:* la jerarquía de excepciones de Java contiene cientos de clases. En la API de Java puede encontrar información acerca de las clases de excepciones de Java. La documentación para la clase `Throwable` se encuentra en `java.sun.com/javase/6/docs/api/java/lang/Throwable.html`. En este sitio puede buscar las subclases de esta clase para obtener más información acerca de los objetos `Exception` y `Error` de Java].

Java clasifica a las excepciones en dos categorías: **excepciones verificadas** y **excepciones no verificadas**. Esta distinción es importante, ya que el compilador de Java implementa un **requerimiento de atrapar o declarar** para las excepciones verificadas. El tipo de una excepción determina si es verificada o no verificada. Todos los tipos de excepciones que son subclases directas o indirectas de la clase **`RuntimeException`** (paquete `java.lang`) son excepciones no verificadas. Esto incluye a las excepciones que ya hemos visto, como las excepciones `ArrayIndexOutOfBoundsException` y `ArithmeticException` (que se muestran en la figura 13.3). Todas las clases que heredan de la clase `Exception` pero no de la clase `RuntimeException` se consideran como excepciones verificadas; y las que heredan de la clase `Error` se consideran como no verificadas. El compilador *verifica* cada una de las llamadas a un método, junto con su declaración, para determinar si el método lanza excepciones verificadas. De ser así, el compilador asegura que la excepción verificada sea atrapada o declarada en una cláusula `throws`. En la sección 13.4 vimos que la cláusula `throws` especifica las excepciones que lanza un método. Dichas excepciones no se atrapan en el cuerpo del método. Para satisfacer la parte relacionada con *atrapar* del requerimiento de atrapar o declarar, el código que genera la excepción debe envolverse en un bloque `try`, y debe proporcionar un manejador `catch` para el tipo de excepción verificada (o uno de los tipos de su superclase). Para satisfacer la parte relacionada con *declarar* del requerimiento de atrapar o declarar, el método que contiene el código que genera la excepción debe proporcionar una cláusula `throws` que contenga el tipo de excepción verificada, después de su lista de parámetros y antes de su cuerpo. Si el requerimiento de atrapar o declarar no se satisface, el compilador emitirá un mensaje de error, indicando que la excepción debe ser atrapada o declarada. Esto obliga a los programadores a pensar acerca de los problemas que pueden ocurrir cuando se hace una llamada a un método que lanza

excepciones verificadas. Las clases de excepciones se definen para verificarse cuando se consideran lo bastante importantes como para atraparlas o declararlas.



Observación de ingeniería de software 13.6

Los programadores se ven obligados a tratar con las excepciones verificadas. Esto produce un código más robusto que el que se crearía si los programadores pudieran simplemente ignorar las excepciones.



Error común de programación 13.4

Si un método intenta de manera explícita lanzar una excepción verificada (o si llama a otro método que lance una excepción verificada), y esa excepción no se lista en la cláusula `throws` *de ese método, se produce un error de compilación.*



Error común de programación 13.5

Si el método de una subclase sobrescribe al método de una superclase, es un error para el método de la subclase listar más expresiones en su cláusula `throws` *de las que tiene el método sobrescrito de la superclase. Sin embargo, la cláusula* `throws` *de una subclase puede contener un subconjunto de la lista* `throws` *de una superclase.*



Observación de ingeniería de software 13.7

Si su método llama a otros métodos que lanzan explícitamente excepciones verificadas, esas excepciones deben atraparse o declararse en su método. Si una expresión puede manejarse de manera significativa en un método, éste debe atrapar la excepción en vez de declararla.

A diferencia de las excepciones verificadas, el compilador de Java no verifica el código para determinar si una excepción no verificada es atrapada o declarada. Por lo general, las excepciones no verificadas se pueden evitar mediante una codificación apropiada. Por ejemplo, la excepción `ArithmeticException` no verificada que lanza el método `cociente` (líneas 9 a 13) en la figura 13.2 puede evitarse si el método se asegura que el denominador no sea cero antes de tratar de realizar la división. No es obligatorio que se listen las excepciones no verificadas en la cláusula `throws` de un método; aun si se listan, no es obligatorio que una aplicación atrape dichas excepciones.



Observación de ingeniería de software 13.8

Aunque el compilador no implementa el requerimiento de atrapar o declarar para las excepciones no verificadas, usted deberá proporcionar un código apropiado para el manejo de excepciones cuando sepa que dichas excepciones podrían ocurrir. Por ejemplo, un programa podría procesar excepciones `NumberFormatException` *del método* `parseInt` *de la clase* `Integer`*, aun cuando las excepciones* `NumberFormatException` *(una subclase de* `RuntimeException`*) sean no verificadas. Esto hará que sus programas sean más robustos.*

Las clases de excepciones se pueden derivar de una superclase común. Si se escribe un manejador `catch` para atrapar objetos de excepción de un tipo de superclase, también se pueden atrapar todos los objetos de las subclases de esa clase. Esto permite que un bloque `catch` maneje los errores relacionados con una notación concisa, y permite el procesamiento polimórfico de las excepciones relacionadas. Evidentemente, se podría atrapar a cada uno de los tipos de las subclases en forma individual, si estas excepciones requirieran un procesamiento distinto. Atrapar excepciones relacionadas en un bloque `catch` tendría sentido solamente si el comportamiento del manejo fuera el mismo para todas las subclases.

Si hay varios bloques `catch` que coinciden con un tipo específico de excepción, sólo reejecuta el primer bloque `catch` que coincida cuando ocurra una excepción de ese tipo. Es un error de compilación tratar de atrapar el mismo tipo exacto en dos bloques `catch` distintos asociados con un bloque `try` específico. Sin embargo, puede haber varios bloques `catch` que coincidan con una excepción; es decir, varios bloques `catch` cuyos tipos sean los mismos que el tipo de excepción, o de una subclase de ese tipo. Por ejemplo, podríamos colocar un bloque `catch` para el tipo `ArithmeticException` después de un bloque `catch` para el tipo `Exception`; ambos coincidirían con las excepciones `ArithmeticException`, pero sólo se ejecutaría el primer bloque `catch` que coincidiera.



Tip para prevenir errores 13.6

Atrapar los tipos de las subclases en forma individual puede ocasionar errores si usted olvida evaluar uno o más de los tipos de subclase en forma explícita; al atrapar a la superclase se garantiza que se atraparán los objetos de todas las subclases. Al colocar un bloque `catch` *para el tipo de la superclase después de los demás bloques* `catch` *para todas las subclases de esa superclase aseguramos que todas las excepciones de las subclases se atrapen en un momento dado.*



Error común de programación 13.6

Al colocar un bloque `catch` *para un tipo de excepción de la superclase antes de los demás bloques* `catch` *que atrapan los tipos de excepciones de las subclases, evitamos que esos bloques* `catch` *se ejecuten, por lo cual se produce un error de compilación.*

13.7 Bloque `finally`

Los programas que obtienen ciertos tipos de recursos deben devolver esos recursos al sistema en forma explícita, para evitar las denominadas **fugas de recursos**. En lenguajes de programación como C y C++, el tipo más común de fuga de recursos es la fuga de memoria. Java realiza la recolección automática de basura en la memoria que ya no es utilizada por los programas, evitando así la mayoría de las fugas de memoria. Sin embargo, pueden ocurrir otros tipos de fugas de recursos en Java. Por ejemplo, los archivos, las conexiones de bases de datos y conexiones de red que no se cierran apropiadamente podrían no estar disponibles para su uso en otros programas.



Tip para prevenir errores 13.7

Hay una pequeña cuestión en Java: no elimina completamente las fugas de memoria. Java no hace recolección de basura en un objeto, sino hasta que no existen más referencias a ese objeto. Por lo tanto, si los programadores mantienen por error referencias a objetos no deseados, pueden ocurrir fugas de memoria.

El bloque `finally` (que consiste en la palabra clave `finally`, seguida de código encerrado entre llaves) es opcional, y algunas veces se le llama **cláusula `finally`**. Si está presente, se coloca después del último bloque `catch`, como en la figura 13.4.

Java garantiza que un bloque `finally` (si hay uno presente en una instrucción `try`) se ejecutará, se lance o no una excepción en el bloque `try` correspondiente, o en cualquiera de sus bloques `catch` correspondientes. Java

```
try
{
    instrucciones
    instrucciones para adquirir recursos
} // fin del bloque try
catch ( UnTipoDeExcepción excepción1 )
{
    instrucciones para manejar excepciones
} // fin de bloque catch
.
.
.
catch ( OtroTipoDeExcepción excepción2 )
{
    instrucciones para manejar excepciones
} // fin de bloque catch
finally
{
    instrucciones
    instrucciones para liberar recursos
} // fin de bloque finally
```

Figura 13.4 | Una instrucción `try` con un bloque `finally`.

también garantiza que un bloque `finally` (si hay uno presente) se ejecutará si un bloque `try` se sale mediante el uso de una instrucción `return`, `break` o `continue`, o simplemente al llegar a la llave derecha de cierre del bloque `try`. El bloque `finally` *no* se ejecutará si la aplicación sale antes de tiempo de un bloque `try`, llamando al método `System.exit`. Este método, que demostraremos en el siguiente capítulo, termina de inmediato una aplicación.

Como un bloque `finally` casi siempre se ejecuta, por lo general, contiene código para liberar recursos. Suponga que se asigna un recurso en un bloque `try`. Si no ocurre una excepción, se ignoran los bloques `catch` y el control pasa al bloque `finally`, que libera el recurso. Después, el control pasa a la primera instrucción después del bloque `finally`. Si ocurre una excepción en el bloque `try`, el programa ignora el resto de este bloque. Si el programa atrapa la excepción en uno de los bloques `catch`, procesa la excepción, después el bloque `finally` libera el recurso y el control pasa a la primera instrucción después del bloque `finally`.



Tip de rendimiento 13.2

Siempre debe liberar cada recurso de manera explícita y lo más pronto posible, una vez que ya no sea necesario. Esto hace que los recursos estén inmediatamente disponibles para que su programa (o cualquier otro programa) los reutilice, con lo cual se mejora la utilización de recursos.



Tip para prevenir errores 13.8

Como se garantiza que el bloque `finally` debe ejecutarse, ocurra o no una excepción en el bloque `try` correspondiente, este bloque es un lugar ideal para liberar los recursos adquiridos en un bloque `try`. Ésta es también una manera efectiva de eliminar las fugas de recursos. Por ejemplo, el bloque `finally` debe cerrar todos los archivos que estén abiertos en el bloque `try`.

Si una excepción que ocurre en un bloque `try` no puede ser atrapada por uno de los manejadores `catch` de ese bloque `try`, el programa ignora el resto del bloque `try` y el control procede al bloque `finally`. Después el programa pasa la excepción al siguiente bloque `try` (por lo general, en el método que hizo la llamada), en donde un bloque `catch` asociado podría atraparla. Este proceso puede ocurrir a través de muchos niveles de bloques `try`. También es posible que la excepción no se atrape.

Si un bloque `catch` lanza una excepción, el bloque `finally` de todas formas se ejecuta. Después, la excepción se pasa al siguiente bloque `try` exterior; de nuevo, en el método que hizo la llamada.

La figura 13.5 demuestra que el bloque `finally` se ejecuta, aun cuando no se lance una excepción en el bloque `try` correspondiente. El programa contiene los métodos `static main` (líneas 7 a 19), `lanzaExcepcion` (líneas 22 a 45) y `noLanzaExcepcion` (líneas 48 a 65). Los métodos `lanzaExcepcion` y `noLanzaExcepcion` se declaran como `static`, por lo que `main` puede llamarlos directamente sin instanciar un objeto `UsoDeExcepciones`.

```
1  // Fig. 13.5: UsoDeExcepciones.java
2  // Demostración del mecanismo de manejo de excepciones
3  // try...catch...finally.
4
5  public class UsoDeExcepciones
6  {
7     public static void main( String args[] )
8     {
9        try
10       {
11          lanzaExcepcion(); //llama al método lanzaExcepcion
12       } // fin de try
13       catch ( Exception excepcion ) // excepción lanzada por lanzaExcepcion
14       {
15          System.err.println( "La excepcion se manejo en main" );
16       } // fin de catch
17
18       noLanzaExcepcion();
19    } // fin de main
```

Figura 13.5 Mecanismo de manejo de excepciones `try...catch...finally`. (Parte 1 de 2).

```java

   // demuestra los bloques try...catch...finally
   public static void lanzaExcepcion() throws Exception
   {
      try // lanza una excepción y la atrapa de inmediato
      {
         System.out.println( "Metodo lanzaExcepcion" );
         throw new Exception(); // genera la excepción
      } // fin de try
      catch ( Exception excepcion ) // atrapa la excepción lanzada en el bloque try
      {
         System.err.println(
            "La excepcion se manejo en el metodo lanzaExcepcion" );
         throw excepcion; // vuelve a lanzar para procesarla más adelante

         // no se llegaría al código que se coloque aquí, la excepción se vuelve a lanzar en el bloque catch

      } // fin de catch
      finally // se ejecuta sin importar lo que ocurra en los bloques try...catch
      {
         System.err.println( "Se ejecuto finally en lanzaExcepcion" );
      } // fin de finally

      // no se llega al código que se coloque aquí, la excepción se vuelve a lanzar en el bloque catch

   } // fin del método lanzaExcepcion

   // demuestra el uso de finally cuando no ocurre una excepción
   public static void noLanzaExcepcion()
   {
      try // el bloque try no lanza una excepción
      {
         System.out.println( "Metodo noLanzaExcepcion" );
      } // fin de try
      catch ( Exception excepcion ) // no se ejecuta
      {
         System.err.println( excepcion );
      } // fin de catch
      finally // se ejecuta sin importar lo que ocurra en los bloques try...catch
      {
         System.err.println(
            "Se ejecuto Finally en noLanzaExcepcion" );
      } // fin de bloque finally

      System.out.println( "Fin del metodo noLanzaExcepcion" );
   } // fin del método noLanzaExcepcion
} // fin de la clase UsoDeExcepciones
```

```
Metodo lanzaExcepcion
La excepcion se manejo en el metodo lanzaExcepcion
Se ejecuto finally en lanzaExcepcion
La excepcion se manejo en main
Metodo noLanzaExcepcion
Se ejecuto Finally en noLanzaExcepcion
Fin del metodo noLanzaExcepcion
```

Figura 13.5 Mecanismo de manejo de excepciones `try`...`catch`...`finally`. (Parte 2 de 2).

Observe el uso de `System.err` para imprimir datos en pantalla (líneas 15, 31–32, 40, 56, 60 y 61). De manera predeterminada, `System.err.println`, al igual que `System.out.println`, muestra los datos en el símbolo del sistema.

Tanto `System.out` como `System.err` son **flujos**: una secuencia de bytes. Mientras que `System.out` (conocido como el **flujo de salida estándar**) se utiliza para mostrar la salida de un programa, `System.err` (conocido como el **flujo de error estándar**) se utiliza para mostrar los errores de un programa. La salida de estos flujos se puede redirigir (es decir, enviar a otra parte que no sea el símbolo del sistema, como a un archivo). El uso de dos flujos distintos permite al programador separar fácilmente los mensajes de error de cualquier otra información de salida. Por ejemplo, los datos que se imprimen de `System.err` se podrían enviar a un archivo de registro, mientras que los que se imprimen de `System.out` se podrían mostrar en la pantalla. Para simplificar, en este capítulo no redimiremos la salida de `System.err`, sino que mostraremos dichos mensajes en el símbolo del sistema. En el capítulo 14, Archivos y flujos, aprenderá más acerca de los flujos.

Lanzar excepciones mediante la instrucción `throw`

El método `main` (figura 13.5) empieza a ejecutarse, entra a su bloque `try` y de inmediato llama al método `lanzaExcepcion` (línea 11). El método `lanzaExcepcion` lanza una excepción tipo `Exception`. La instrucción en la línea 27 se conoce como **instrucción `throw`**; esta instrucción se ejecuta para indicar que ha ocurrido una excepción. Hasta ahora sólo hemos atrapado las excepciones que lanzan los métodos que son llamados. Los programadores pueden lanzar excepciones mediante el uso de la instrucción `throw`. Al igual que con las excepciones lanzadas por los métodos de la API de Java, esto indica a las aplicaciones cliente que ha ocurrido un error. Una instrucción `throw` especifica un objeto que se lanzará. El operando de `throw` puede ser de cualquier clase derivada de `Throwable`.



Observación de ingeniería de software 13.9

Cuando se invoca el método `toString` en cualquier objeto `Throwable`, su cadena resultante incluye la cadena descriptiva que se suministró al constructor, o simplemente el nombre, si no se suministró una cadena.



Observación de ingeniería de software 13.10

Un objeto puede lanzarse sin contener información acerca del problema que ocurrió. En este caso, el simple conocimiento de que ocurrió una excepción de cierto tipo puede proporcionar suficiente información para que el manejador procese el problema en forma correcta.



Observación de ingeniería de software 13.11

Las excepciones pueden lanzarse desde constructores. Cuando se detecta un error en un constructor, debe lanzarse una excepción en vez de crear un objeto formado en forma inapropiada.

Volver a lanzar excepciones

La línea 33 de la figura 13.5 **vuelve a lanzar la excepción**. Las excepciones se vuelven a lanzar cuando un bloque `catch`, al momento de recibir una excepción, decide que no puede procesar la excepción o que sólo puede procesarla parcialmente. Al volver a lanzar una excepción, se difiere el manejo de la misma (o tal vez una porción de ella) hacia otro bloque `catch` asociado con una instrucción `try` exterior. Para volver a lanzar una excepción se utiliza la **palabra clave `throw`**, seguida de una referencia al objeto excepción que se acaba de atrapar. Observe que las excepciones no se pueden volver a lanzar desde un bloque `finally`, ya que el parámetro de la excepción del bloque `catch` ha expirado.

Cuando se vuelve a lanzar una excepción, el siguiente bloque `try` circundante la detecta, y la instrucción `catch` de ese bloque `try` trata de manejarla. En este caso, el siguiente bloque `try` circundante se encuentra en las líneas 9 a 12 en el método `main`. Sin embargo, antes de manejar la excepción que se volvió a lanzar, se ejecuta el bloque `finally` (líneas 38 a 41). Después, el método `main` detecta la excepción que se volvió a lanzar en el bloque `try`, y la maneja en el bloque `catch` (líneas 13 a 16).

A continuación, `main` llama al método `noLanzaExcepcion` (línea 18). Como no se lanza una excepción en el bloque `try` de `noLanzaExcepcion` (líneas 50 a 53), el programa ignora el bloque `catch` (líneas 54 a 57), pero el bloque `finally` (líneas 58 a 62) se ejecuta de todas formas. El control pasa a la instrucción que está después del bloque `finally` (línea 64). Después, el control regresa a `main` y el programa termina.



Error común de programación 13.7

Si no se ha atrapado una excepción cuando el control entra a un bloque `finally`, y éste lanza una excepción que no se atrapa en el bloque `finally`, se perderá la primera excepción y se devolverá la excepción del bloque `finally` al método que hizo la llamada.



Tip para prevenir errores 13.9

Evite colocar código que pueda lanzar (`throw`) una excepción en un bloque `finally`. Si se requiere dicho código, enciérrelo en bloques `try...catch` dentro del bloque `finally`.



Error común de programación 13.8

Suponer que una excepción lanzada desde un bloque `catch` se procesará por ese bloque `catch`, o por cualquier otro bloque `catch` asociado con la misma instrucción `try`, puede provocar errores lógicos.



Buena práctica de programación 13.2

El mecanismo de manejo de excepciones de Java está diseñado para eliminar el código de procesamiento de errores de la línea principal del código de un programa, para así mejorar su legibilidad. No coloque bloques `try...catch...finally` alrededor de cada instrucción que pueda lanzar una excepción. Esto dificulta la legibilidad de los programas. En vez de ello, coloque un bloque `try` alrededor de una porción considerable de su código, y después de ese bloque `try` coloque bloques `catch` para manejar cada posible excepción, y después de esos bloques `catch` coloque un solo bloque `finally` (si se requiere).

13.8 Limpieza de la pila

Cuando se lanza una excepción, pero no se atrapa en un alcance específico, la pila de llamadas a métodos se "limpia" y se hace un intento de atrapar (`catch`) la excepción en el siguiente bloque `try` exterior. A este proceso se le conoce como **limpieza de la pila**. Limpiar la pila de llamadas a métodos significa que el método en el que no se atrapó la excepción termina, todas las variables en ese método quedan fuera de alcance y el control regresa a la instrucción que invocó originalmente a ese método. Si un bloque `try` encierra a esa instrucción, se hace un intento de atrapar (`catch`) esa excepción. Si un bloque `try` no encierra a esa instrucción, se lleva a cabo la limpieza de la pila otra vez. Si ningún bloque `catch` atrapa a esta excepción, y la excepción es verificada (como en el siguiente ejemplo), al compilar el programa se producirá un error. El programa de la figura 13.6 demuestra la limpieza de la pila.

```java
// Fig. 13.6: UsoDeExcepciones.java
// Demostración de la limpieza de la pila.

public class UsoDeExcepciones
{
   public static void main( String args[] )
   {
      try // llama a lanzaExcepcion para demostrar la limpieza de la pila
      {
         lanzaExcepcion();
      } // fin de try
      catch ( Exception excepcion ) // excepción lanzada en lanzaExcepcion
      {
         System.err.println( "La excepcion se manejo en main" );
      } // fin de catch
   } // fin de main

   // lanzaExcepcion lanza la excepción que no se atrapa en este método
   public static void lanzaExcepcion() throws Exception
```

Figura 13.6 | Limpieza de la pila. (Parte 1 de 2).

```
20       {
21          try // lanza una excepción y la atrapa en main
22          {
23             System.out.println( "Metodo lanzaExcepcion" );
24             throw new Exception(); // genera la excepción
25          } // fin de try
26          catch ( RuntimeException runtimeException ) // atrapa el tipo incorrecto
27          {
28             System.err.println(
29                "La excepcion se manejo en el metodo lanzaExcepcion" );
30          } // fin de catch
31          finally // el bloque finally siempre se ejecuta
32          {
33             System.err.println( "Finally siempre se ejecuta" );
34          } // fin de finally
35       } // fin del método lanzaExcepcion
36    } // fin de la clase UsoDeExcepciones
```

```
Metodo lanzaExcepcion
Finally siempre se ejecuta
La excepcion se manejo en main
```

Figura 13.6 | Limpieza de la pila. (Parte 2 de 2).

Cuando se ejecuta el método `main`, la línea 10 en el bloque `try` llama al método `lanzaExcepcion` (líneas 19 a 35). En el bloque `try` del método `lanzaExcepcion` (líneas 21 a 25), la línea 24 lanza una excepción `Exception`. Esto termina el bloque `try` de inmediato, y el control ignora el bloque `catch` en la línea 26, debido a que el tipo que se está atrapando (`RuntimeException`) no es una coincidencia exacta con el tipo lanzado (`Exception`) y no es una superclase del mismo. El método `lanzaExcepcion` termina (pero no hasta que se ejecute su bloque `Finally`) y devuelve el control a la línea 10; el punto desde el cual se llamó en el programa. La línea 10 es un bloque `try` circundante. La excepción no se ha manejado todavía, por lo que el bloque `try` termina y se hace un intento por atrapar la excepción en la línea 12. El tipo que se atrapará (`Exception`) no coincide con el tipo lanzado. En consecuencia, el bloque `catch` procesa la excepción y el programa termina al final de `main`. Si no hubiera bloques `catch` que coincidieran, se produciría un error de compilación. Recuerde que éste no es siempre el caso; para las excepciones no verificadas la aplicación se compilará, pero se ejecutará con resultados inesperados.

13.9 printStackTrace, getStackTrace y getMessage

En la sección 13.6 vimos que las excepciones se derivan de la clase `Throwable`. Esta clase ofrece un método llamado **`printStackTrace`**, que envía al flujo de error estándar la pila de llamadas a métodos (lo cual se describe en la sección 13.3). A menudo, esto ayuda en la prueba y la depuración. La clase `Throwable` también proporciona un método llamado **`getStackTrace`**, que obtiene la información de rastreo de la pila que podría imprimir `printStackTrace`. El método **`getMessage`** de la clase `Throwable` devuelve la cadena descriptiva almacenada en una excepción. El ejemplo de esta sección, considera estos tres métodos.



Tip para prevenir errores 13.10

Una excepción que no sea atrapada en una aplicación hará que se ejecute el manejador de excepciones predeterminado de Java. Éste muestra el nombre de la excepción, un mensaje descriptivo que indica el problema que ocurrió y un rastreo completo de la pila de ejecución. En una aplicación con un solo subproceso de ejecución, la aplicación termina. En una aplicación con varios subprocesos, termina el subproceso que produjo la excepción.



Tip para prevenir errores 13.11

El método `toString` de `Throwable` (heredado en todas las subclases de `Throwable`) devuelve una cadena que contiene el nombre de la clase de la excepción y un mensaje descriptivo.

En la figura 13.7 se demuestra el uso de `getMessage`, `printStackTrace` y `getStackTrace`. Si queremos mostrar la información de rastreo de la pila a flujos que no sean el flujo de error estándar, podemos utilizar la información devuelta por `getStackTrace` y enviar estos datos a otro flujo. En el capítulo 14, Archivos y flujos, veremos cómo enviar datos a otros flujos.

En `main`, el bloque `try` (líneas 8 a 11) llama a `metodo1` (declarado en las líneas 35 a 38). A continuación, `metodo1` llama a `metodo2` (declarado en las líneas 41 a 44), que a su vez llama a `metodo3` (declarado en las líneas 47 a 50). En la línea 49 de `metodo3` se lanza un objeto `Exception`; éste es el punto de lanzamiento. Como la instrucción `throw` de la línea 49 no va encerrada en ningún bloque `try`, se lleva a cabo la limpieza de la pila; `metodo3` termina en la línea 49 y después regresa el control a la instrucción en `metodo2` que invocó a `metodo3` (es decir, la línea 43). Como ningún bloque `try` encierra a la línea 43, se lleva a cabo la limpieza de la pila otra vez; `metodo2` termina en la línea 43 y regresa el control a la instrucción en `metodo1` que invocó a `metodo2` (es decir, la línea 37). Como ningún bloque `try` encierra a la línea 37, se lleva a cabo la limpieza de la pila una vez más; `metodo1` termina en la línea 37 y regresa el control a la instrucción en `main` que invocó a `metodo1` (es decir, la línea 10). El bloque `try` de las líneas 8 a 11 encierra a esta instrucción. La excepción no ha sido manejada, por lo que el bloque `try` termina y el primer bloque `catch` concordante (líneas 12 a 31) atrapa y procesa la excepción.

```
1  // Fig. 13.7: UsoDeExcepciones.java
2  // Demostración de getMessage y printStackTrace de la clase Exception.
3
4  public class UsoDeExcepciones
5  {
6     public static void main( String args[] )
7     {
8        try
9        {
10          metodo1(); // llama a metodo1
11       } // fin de try
12       catch ( Exception excepcion ) // atrapa la excepción lanzada en metodo1
13       {
14          System.err.printf( "%s\n\n", excepcion.getMessage() );
15          excepcion.printStackTrace(); // imprime el rastreo de la pila de la excepción
16
17          // obtiene la información de rastreo de la pila
18          StackTraceElement[] elementosRastreo = excepcion.getStackTrace();
19
20          System.out.println( "\nRastreo de la pila de getStackTrace:" );
21          System.out.println( "Clase\t\t\tArchivo\t\t\tLinea\tMetodo" );
22
23          // itera a través de elementosRastreo para obtener la descripción de la
               excepción
24          for ( StackTraceElement elemento : elementosRastreo )
25          {
26             System.out.printf( "%s\t", elemento.getClassName() );
27             System.out.printf( "%s\t", elemento.getFileName() );
28             System.out.printf( "%s\t", elemento.getLineNumber() );
29             System.out.printf( "%s\n", elemento.getMethodName() );
30          } // fin de for
31       } // fin de catch
32    } // fin de main
33
34    // llama a metodo2; lanza las excepciones de vuelta a main
35    public static void metodo1() throws Exception
36    {
37       metodo2();
38    } // fin del método metodo1
39
```

Figura 13.7 | Los métodos `getMessage`, `getStackTrace` y `printStackTrace` de `Throwable`. (Parte 1 de 2).

```
40     // llama a metodo3; lanza las excepciones de vuelta a metodo1
41     public static void metodo2() throws Exception
42     {
43        metodo3();
44     } // fin del método metodo2
45
46     // lanza la excepción Exception de vuelta a metodo2
47     public static void metodo3() throws Exception
48     {
49        throw new Exception( "La excepcion se lanzo en metodo3" );
50     } // fin del método metodo3
51  } // fin de la clase UsoDeExcepciones
```

```
La excepcion se lanzo en metodo3

java.lang.Exception: La excepcion se lanzo en metodo3
        at UsoDeExcepciones.metodo3(UsoDeExcepciones.java:49)
        at UsoDeExcepciones.metodo2(UsoDeExcepciones.java:43)
        at UsoDeExcepciones.metodo1(UsoDeExcepciones.java:37)
        at UsoDeExcepciones.main(UsoDeExcepciones.java:10)

Rastreo de la pila de getStackTrace:
Clase              Archivo                   Linea   Metodo
UsoDeExcepciones   UsoDeExcepciones.java     49      metodo3
UsoDeExcepciones   UsoDeExcepciones.java     43      metodo2
UsoDeExcepciones   UsoDeExcepciones.java     37      metodo1
UsoDeExcepciones   UsoDeExcepciones.java     10      main
```

Figura 13.7 | Los métodos `getMessage`, `getStackTrace` y `printStackTrace` de `Throwable`. (Parte 2 de 2).

En la línea 14 se invoca al método `getMessage` de la excepción, para obtener la descripción de la misma. En la línea 15 se invoca al método `printStackTrace` de la excepción, para mostrar el rastreo de la pila, el cual indica en dónde ocurrió la excepción. En la línea 18 se invoca al método `getStackTrace` de la excepción, para obtener la información del rastreo de la pila, como un arreglo de objetos **`StackTraceElement`**. En las líneas 24 a 30 se obtiene cada uno de los objetos `StackTraceElement` en el arreglo, y se invocan sus métodos **`getClassName`**, **`getFileName`**, **`getLineNumber`** y **`getMethodName`** para obtener el nombre de la clase, el nombre del archivo, el número de línea y el nombre del método, respectivamente, para ese objeto `StackTraceElement`. Cada objeto `StackTraceElement` representa la llamada a un método en la pila de llamadas a métodos.

Los resultados de la figura 13.7 muestran que la información de rastreo de la pila que imprime `printStackTrace` sigue el patrón: *nombreClase.nombreMétodo(nombreArchivo:númeroLínea)*, en donde *nombreClase*, *nombreMétodo* y *nombreArchivo* indican los nombres de la clase, el método y el archivo en los que ocurrió la excepción, respectivamente, y *númeroLínea* indica en qué parte del archivo ocurrió la excepción. Usted vio esto en los resultados para la figura 13.1. El método `getStackTrace` permite un procesamiento personalizado de la información sobre la excepción. Compare la salida de `printStackTrace` con la salida creada a partir de los objetos `StackTraceElement`, y podrá ver que ambos contienen la misma información de rastreo de la pila.



Observación de ingeniería de software 13.12

Nunca ignore una excepción que atrape. Por lo menos, use el método `printStackTrace` para imprimir un mensaje de error. Esto informará a los usuarios que existe un problema, para que puedan tomar las acciones apropiadas.

13.10 Excepciones encadenadas

Algunas veces un bloque `catch` atrapa un tipo de excepción y después lanza una nueva excepción de un tipo distinto, para indicar que ocurrió una excepción específica del programa. En las primeras versiones de Java, no había mecanismo para envolver la información de la excepción original con la información de la nueva excepción, para proporcionar un rastreo completo de la pila, indicando en dónde ocurrió el problema original en el programa. Esto hacía que depurar dichos problemas fuera un proceso bastante difícil. Las **excepciones encadenadas** permiten que un objeto de excepción mantenga la información completa sobre el rastreo de la pila. En la figura 13.8 se demuestran las excepciones encadenadas.

```
// Fig. 13.8: UsoDeExcepcionesEncadenadas.java
// Demostración de las excepciones encadenadas.

public class UsoDeExcepcionesEncadenadas
{
   public static void main( String args[] )
   {
      try
      {
         metodo1(); // llama a metodo1
      } // fin de try
      catch ( Exception excepcion ) // excepciones lanzadas desde metodo1
      {
         excepcion.printStackTrace();
      } // fin de catch
   } // fin de main

   // llama a metodo2; lanza las excepciones de vuelta a main
   public static void metodo1() throws Exception
   {
      try
      {
         metodo2(); // llama a metodo2
      } // fin de try
      catch ( Exception excepcion ) // excepción lanzada desde metodo2
      {
         throw new Exception( "La excepcion se lanzo en metodo1", excepcion );
      } // fin de try
   } // fin del método metodo1

   // llama a metodo3; lanza las excepciones de vuelta a metodo1
   public static void metodo2() throws Exception
   {
      try
      {
         metodo3(); // llama a metodo3
      } // fin de try
      catch ( Exception excepcion ) // excepción lanzada desde metodo3
      {
         throw new Exception( "La excepcion se lanzo en metodo2", excepcion );
      } // fin de catch
   } // fin del método metodo2

   // lanza excepción Exception de vuelta a metodo2
   public static void metodo3() throws Exception
   {
      throw new Exception( "La excepcion se lanzo en metodo3" );
   } // fin del método metodo3
} // fin de la clase UsoDeExcepcionesEncadenadas
```

Figura 13.8 | Excepciones encadenadas. (Parte 1 de 2).

```
java.lang.Exception: La excepcion se lanzo en metodo1
        at UsoDeExcepcionesEncadenadas.metodo1(UsoDeExcepcionesEncadenadas.java:27)
        at UsoDeExcepcionesEncadenadas.main(UsoDeExcepcionesEncadenadas.java:10)
Caused by: java.lang.Exception: La excepcion se lanzo en metodo2
        at UsoDeExcepcionesEncadenadas.metodo2(UsoDeExcepcionesEncadenadas.java:40)
        at UsoDeExcepcionesEncadenadas.metodo1(UsoDeExcepcionesEncadenadas.java:23)
        ... 1 more
Caused by: java.lang.Exception: La excepcion se lanzo en metodo3
        at UsoDeExcepcionesEncadenadas.metodo3(UsoDeExcepcionesEncadenadas.java:47)
        at UsoDeExcepcionesEncadenadas.metodo2(UsoDeExcepcionesEncadenadas.java:36)
        ... 2 more
```

Figura 13.8 | Excepciones encadenadas. (Parte 2 de 2).

El programa consiste de cuatro métodos: `main` (líneas 6 a 16), `metodo1` (líneas 19 a 29), `metodo2` (líneas 32 a 42) y `metodo3` (líneas 45 a 48). La línea 10 en el bloque `try` de `main` llama a `metodo1`. La línea 23 en el bloque `try` de `metodo1` llama a `metodo2`. La línea 36 en el bloque `try` de `metodo2` llama a `metodo3`. En `metodo3`, la línea 47 lanza una nueva excepción `Exception`. Como esta instrucción no se encuentra dentro de un bloque `try`, el `metodo3` termina y la excepción se devuelve al método que hace la llamada (`metodo2`), en la línea 36. Esta instrucción *se encuentra* dentro de un bloque `try`; por lo tanto, el bloque `try` termina y la excepción es atrapada en las líneas 38 a 41. En la línea 40, en el bloque `catch`, se lanza una nueva excepción. En este caso, se hace una llamada al constructor `Exception` con dos argumentos). El segundo argumento representa a la excepción que era la causa original del problema. En este programa, la excepción ocurrió en la línea 47. Como se lanza una excepción desde el bloque `catch`, el `metodo2` termina y devuelve la nueva excepción al método que hace la llamada (`metodo1`), en la línea 23. Una vez más, esta instrucción se encuentra dentro de un bloque `try`, por lo tanto, este bloque termina y la excepción es atrapada en las líneas 25 a 28. En la línea 27, en el bloque `catch` se lanza una nueva excepción y se utiliza la excepción que se atrapó como el segundo argumento para el constructor de `Exception`. Como se lanza una excepción desde el bloque `catch`, el `metodo1` termina y devuelve la nueva excepción al método que hace la llamada (`main`), en la línea 10. El bloque `try` en `main` termina y la excepción es atrapada en las líneas 12 a 15. En la línea 14 se imprime un rastreo de la pila.

Observe en la salida del programa que las primeras tres líneas muestran la excepción más reciente que fue lanzada (es decir, la del `metodo1` en la línea 23). Las siguientes cuatro líneas indican la excepción que se lanzó desde el `metodo2`, en la línea 40. Por último, las siguientes cuatro líneas representan la excepción que se lanzó desde el `metodo3`, en la línea 47. Además observe que, si lee la salida en forma inversa, muestra cuántas excepciones encadenadas más quedan pendientes.

13.11 Declaración de nuevos tipos de excepciones

La mayoría de los programadores de Java utilizan las clases existentes de la API de Java, de distribuidores independientes y de bibliotecas de clases gratuitas (que, por lo general, se pueden descargar de Internet) para crear aplicaciones de Java. Los métodos de esas clases por lo general se declaran para lanzar las excepciones apropiadas cuando ocurren problemas. Los programadores escriben código para procesar esas excepciones existentes, para que sus programas sean más robustos.

Si usted crea clases que otros programadores utilizarán en sus programas, tal vez le sea conveniente declarar sus propias clases de excepciones que sean específicas para los problemas que pueden ocurrir cuando otro programador utilice sus clases reutilizables.



Observación de ingeniería de software 13.13

De ser posible, indique las excepciones de sus métodos mediante el uso de las clases de excepciones existentes, en vez de crear nuevas clases de excepciones. La API de Java contiene muchas clases de excepciones que podrían ser adecuadas para el tipo de problema que su método necesite indicar.

Una nueva clase de excepción debe extender a una clase de excepción existente, para poder asegurar que la clase pueda utilizarse con el mecanismo de manejo de excepciones. Al igual que cualquier otra clase, una clase de excepción puede contener campos y métodos. Sin embargo, una nueva clase de excepción, por lo general, contiene sólo dos constructores; uno que no toma argumentos y pasa un mensaje de excepción predeterminado al constructor de la superclase, y otro que recibe un mensaje de excepción personalizado como una cadena y lo pasa al constructor de la superclase.



Buena práctica de programación 13.3

Asociar cada uno de los tipos de fallas graves en tiempo de ejecución con una clase de excepción con nombre apropiado ayuda a mejorar la claridad del programa.



Observación de ingeniería de software 13.14

Al definir su propio tipo de excepción, estudie las clases de excepción existentes en la API de Java y trate de extender una clase de excepción relacionada. Por ejemplo, si va a crear una nueva clase para representar cuando un método intenta realizar una división entre cero, podría extender la clase `ArithmeticException`, ya que la división entre cero ocurre durante la aritmética. Si las clases existentes no son superclases apropiadas para su nueva clase de excepción, debe decidir si su nueva clase debe ser una clase de excepción verificada o no verificada. La nueva clase de excepción debe ser una excepción verificada (es decir, debe extender a `Exception` pero no a `RuntimeException`) los posibles clientes deben manejar la excepción. La aplicación cliente debe ser capaz de recuperarse en forma razonable de una excepción de este tipo. La nueva clase de excepción debe extender a `RuntimeExcepcion` si el código cliente debe ser capaz de ignorar la excepción (es decir, si la excepción es una excepción no verificada).

En el capítulo 17, Estructuras de datos, proporcionaremos un ejemplo de una clase de excepción personalizada. Declararemos una clase reutilizable llamada `Lista`, la cual es capaz de almacenar una lista de referencias a objetos. Algunas operaciones que se realizan comúnmente en una `Lista` no se permitirán si la `Lista` está vacía, como eliminar un elemento de la parte frontal o posterior de la lista (es decir, no pueden eliminarse elementos, ya que la `Lista` no contiene ningún elemento en ese momento). Por esta razón, algunos métodos de `Lista` lanzan excepciones de la clase de excepción `ListaVaciaException`.



Buena práctica de programación 13.4

Por convención, todos los nombres de las clases de excepciones deben terminar con la palabra `Exception`.

13.12 Precondiciones y poscondiciones

Los programadores invierten una gran parte de su tiempo en mantener y depurar código. Para facilitar estas tareas y mejorar el diseño en general, comúnmente especifican los estados esperados antes y después de la ejecución de un método. A estos estados se les llama precondiciones y poscondiciones, respectivamente.

Una **precondición** debe ser verdadera cuando se invoca a un método. Las precondiciones describen las restricciones en los parámetros de un método, y en cualquier otra expectativa que tenga el método en relación con el estado actual de un programa. Si no se cumplen las precondiciones, entonces el comportamiento del método es indefinido; puede lanzar una excepción, continuar con un valor ilegal o tratar de recuperarse del error. Sin embargo, nunca hay que confiar en las precondiciones o esperar un comportamiento consistente, si éstas no se cumplen.

Una **poscondición** es verdadera una vez que el método regresa con éxito. Las poscondiciones describen las restricciones en el valor de retorno, y en cualquier otro efecto secundario que pueda tener el método. Al llamar a un método, podemos asumir que éste satisface todas sus poscondiciones. Si usted está escribiendo su propio método, debe documentar todas las poscondiciones, de manera que otros sepan qué pueden esperar al llamar a su método, y usted debe asegurarse que su método cumpla con todas sus poscondiciones, si en definitiva se cumplen sus precondiciones.

Cuando no se cumplen sus precondiciones o poscondiciones, los métodos por lo general lanzan excepciones. Como ejemplo, examine el método `charAt` de `String`, que tiene un parámetro `int`: un índice en el objeto `String`. Para una precondición, el método `charAt` asume que `indice` es mayor o igual a cero, y menor que la

longitud del objeto `String`. Si se cumple la precondición, ésta establece que el método devolverá el carácter en la posición en el objeto `String` especificada por el parámetro `indice`. En caso contrario, el método lanza una excepción `IndexOutOfBoundsException`. Confiamos en que el método `charAt` satisfaga su poscondición, siempre y cuando cumplamos con la precondición. No necesitamos preocuparnos por los detalles acerca de cómo el método obtiene en realidad el carácter en el índice.

Algunos programadores establecen las precondiciones y poscondiciones de manera informal, como parte de la especificación general del método, mientras que otros prefieren un enfoque más formal, al definirlas de manera explícita. Al diseñar sus propios métodos, usted debe establecer las precondiciones y poscondiciones en un comentario antes de la declaración del método, de cualquier forma que guste. Establecer las precondiciones y poscondiciones antes de escribir un método también nos ayuda a guiarnos a medida que implementamos el método.

13.13 Aserciones

Al implementar y depurar una clase, algunas veces es conveniente establecer condiciones que deban ser verdaderas en un punto específico de un método. Estas condiciones, conocidas como **aserciones**, ayudan a asegurar la validez de un programa al atrapar los errores potenciales e identificar los posibles errores lógicos durante el desarrollo. Las precondiciones y las poscondiciones son dos tipos de aserciones. Las precondiciones son aserciones acerca del estado de un programa a la hora de invocar un método, y las poscondiciones son aserciones acerca del estado de un programa cuando el método termina.

Aunque las aserciones pueden establecerse como comentarios para guiar al programador durante el desarrollo, Java incluye dos versiones de la instrucción **`assert`** para validar aserciones mediante la programación. La instrucción `assert` evalúa una expresión `bolean` y determina si es verdadera o falsa. La primera forma de la instrucción `assert` es

`assert` *expresión*;

Esta instrucción evalúa *expresión* y lanza una excepción `AssertionError` si la expresión es `falsa`. La segunda forma es

`assert` *expresión1* `:` *expresión2*;

Esta instrucción evalúa *expresión1* y lanza una excepción `AssertionError` con *expresión2* como el mensaje de error, en caso de que *expresión1* sea `false`.

Puede utilizar aserciones para implementar las precondiciones y poscondiciones mediante la programación, o para verificar cualquier otro estado intermedio que le ayude a asegurar que su código esté funcionando en forma correcta. El ejemplo de la figura 13.9 demuestra la funcionalidad de la instrucción `assert`. En la línea 11 se pide al usuario que introduzca un número entre 0 y 10, y después en la línea 12 se lee el número de la línea de comandos. La instrucción `assert` en la línea 15 determina si el usuario introdujo un número dentro del rango válido. Si el número está fuera de rango, entonces el programa reporta un error; en caso contrario, continúa en forma normal.

```
1  // Fig. 13.9: PruebaAssert.java
2  // Uso de assert para verificar que un valor absoluto sea positivo
3  import java.util.Scanner;
4
5  public class PruebaAssert
6  {
7     public static void main( String args[] )
8     {
9        Scanner entrada = new Scanner( System.in );
10
11       System.out.print( "Escriba un numero entre 0 y 10: " );
12       int numero = entrada.nextInt();
13
14       // asegura que el valor absoluto sea >= 0
15       assert ( numero >= 0 && numero <= 10 ) : "numero incorrecto: " + numero;
```

Figura 13.9 | Verificar con `assert` que un valor se encuentre dentro del rango. (Parte 1 de 2).

```

      System.out.printf( "Usted escribio %d\n", numero );
   } // fin de main
} // fin de la clase PruebaAssert
```

```
Escriba un numero entre 0 y 10: 5
Usted escribio 5
```

```
Escriba un numero entre 0 y 10: 50
Exception in thread "main" java.lang.AssertionError: numero incorrecto: 50
        at PruebaAssert.main(PruebaAssert.java:15)
```

Figura 13.9 | Verificar con `assert` que un valor se encuentre dentro del rango. (Parte 2 de 2).

El programador utiliza las aserciones principalmente para depurar e identificar errores lógicos en una aplicación. De manera predeterminada, las aserciones están deshabilitadas al ejecutar un programa, ya que reducen el rendimiento y son innecesarias para el usuario del programa. Para habilitar las aserciones en tiempo de ejecución, use la opción de línea de comandos `-ea` del comando `java`. Para ejecutar el programa de la figura 13.9 con las aserciones habilitadas, escriba

```
java -ea PruebaAssert
```

No debe encontrar ningún error tipo `AssertionError` durante la ejecución normal de un programa escrito en forma apropiada. Dichos errores sólo deben indicar errores en la implementación. Como resultado, nunca se debe atrapar una excepción tipo `AssertionError`. En vez de ello, debemos permitir que el programa termine al ocurrir el error, para poder ver el mensaje de error; después hay que localizar y corregir el origen del problema. Como los usuarios de las aplicaciones pueden elegir no habilitar las aserciones en tiempo de ejecución, no debemos usar la instrucción `assert` para indicar problemas en tiempo de ejecución en el código de producción. En vez de ello, debemos usar el mecanismo de las excepciones para este fin.

13.14 Conclusión

En este capítulo aprendió a utilizar el manejo de excepciones para lidiar con los errores en una aplicación. Aprendió que el manejo de excepciones permite a los programadores eliminar el código para manejar errores de la "línea principal" de ejecución del programa. Vio el manejo de errores en el contexto de un ejemplo de división entre cero. Aprendió a utilizar los bloques `try` para encerrar código que puede lanzar una excepción, y cómo utilizar los bloques `catch` para lidiar con las excepciones que puedan surgir. Aprendió acerca del modelo de terminación del manejo de excepciones, que indica que una vez que se maneja una excepción, el control del programa no regresa al punto de lanzamiento. Conoció la diferencia entre las excepciones verificadas y no verificadas, y cómo especificar mediante la cláusula `throws` que las excepciones específicas que ocurran en un método serán lanzadas por ese método al método que lo llamó. Aprendió a utilizar el bloque `finally` para liberar recursos, ya sea que ocurra o no una excepción. También aprendió a lanzar y volver a lanzar excepciones. Después, aprendió a obtener información acerca de una excepción, mediante el uso de los métodos `printStackTrace`, `getStackTrace` y `getMessage`. El capítulo continuó con una discusión sobre las excepciones encadenadas, que permiten a los programadores envolver la información de la excepción original con la información de la nueva excepción. Después, vimos las generalidades acerca de cómo crear sus propias clases de excepciones. Presentamos las precondiciones y poscondiciones para ayudar a los programadores que utilizan sus métodos a comprender las condiciones que deben ser verdaderas cuando se hace la llamada al método, y cuando éste regresa. Cuando no se cumplen las precondiciones y poscondiciones, los métodos generalmente lanzan excepciones. Por último, hablamos sobre la instrucción `assert` y cómo puede utilizarse para ayudarnos a depurar los programas. En especial, esta instrucción se puede utilizar para asegurar que se cumplan las precondiciones y poscondiciones. En el siguiente capítulo aprenderá acerca del procesamiento de archivos, incluyendo la forma en que se almacenan los datos persistentes y cómo se manipulan.

Resumen

Sección 13.1 Introducción

- Una excepción es una indicación de un problema que ocurre durante la ejecución de un programa.
- El manejo de excepciones permite a los programadores crear aplicaciones que puedan resolver las excepciones.

Sección 13.2 Generalidades acerca del manejo de excepciones

- El manejo de excepciones permite a los programadores eliminar el código para manejar errores de la "línea principal" de ejecución del programa, mejorando su claridad y capacidad de modificación.

Sección 13.3 Ejemplo: división entre cero sin manejo de excepciones

- Las excepciones se lanzan cuando un método detecta un problema y no puede manejarlo.
- El rastreo de la pila de una excepción incluye el nombre de la excepción en un mensaje descriptivo, el cual indica el problema que ocurrió y la pila de llamadas a métodos completa (es decir, la cadena de llamadas), en el momento en el que ocurrió la excepción.
- El punto en el programa en el cual ocurre una excepción se conoce como punto de lanzamiento.

Sección 13.4 Ejemplo: manejo de excepciones tipo `ArithmeticException` e `InputMismatchException`

- Un bloque `try` encierra el código que podría lanzar una excepción, y el código que no debe ejecutarse si se produce esa excepción.
- Las excepciones pueden surgir a través de código mencionado explícitamente en un bloque `try`, a través de llamadas a otros métodos, o incluso a través de llamadas a métodos anidados, iniciadas por el código en el bloque `try`.
- Un bloque `catch` empieza con la palabra clave `catch` y un parámetro de excepción, seguido de un bloque de código que atrapa (es decir, recibe) y maneja la excepción. Este código se ejecuta cuando el bloque `try` detecta la excepción.
- Una excepción no atrapada es una excepción que ocurre y para la cual no hay bloques `catch` que coincidan.
- Una excepción no atrapada hará que un programa termine antes de tiempo, si éste sólo contiene un subproceso. Si el programa contiene más de un subproceso, sólo terminará el subproceso en el que ocurrió la excepción. El resto del programa se ejecutará, pero puede producir efectos adversos.
- Justo después del bloque `try` debe ir por lo menos un bloque `catch` o un bloque `finally`.
- Cada bloque `catch` especifica entre paréntesis un parámetro de excepción, el cual identifica el tipo de excepción que puede procesar el manejador. El nombre del parámetro de excepción permite al bloque `catch` interactuar con un objeto de excepción atrapada.
- Si ocurre una excepción en un bloque `try`, éste termina de inmediato y el control del programa se transfiere al primero de los siguientes bloques `catch` cuyo parámetro de excepción coincida con el tipo de la excepción que se lanzó.
- Una vez que se maneja una excepción, el control del programa no regresa al punto de lanzamiento, ya que el bloque `try` ha expirado. A esto se le conoce como el modelo de terminación del manejo de excepciones.
- Si hay varios bloques `catch` que coinciden cuando ocurre una excepción, sólo se ejecuta el primero.
- Después de ejecutar un bloque `catch`, el flujo de control del programa pasa a la siguiente instrucción después del último bloque `catch`.
- Una cláusula `throws` especifica las excepciones que lanza el método, y aparece después de la lista de parámetros del método, pero antes de su cuerpo.
- La cláusula `throws` contiene una lista separada por comas de excepciones que lanzará el método, en caso de que ocurra un problema cuando el método se ejecute.

Sección 13.5 Cuándo utilizar el manejo de excepciones

- El manejo de excepciones está diseñado para procesar errores sincrónicos, que ocurren cuando se ejecuta una instrucción.
- El manejo de excepciones no está diseñado para procesar los problemas asociados con eventos asíncronos, que ocurren en paralelo con (y en forma independiente de) el flujo de control del programa.

Sección 13.6 Jerarquía de excepciones de Java

- Todas las clases de excepciones de Java heredan, ya sea en forma directa o indirecta, de la clase `Exception`. Debido a esto, las clases de excepciones de Java forman una jerarquía. Los programadores pueden extender esta jerarquía para crear sus propias clases de excepciones.

- La clase `Throwable` es la superclase de la clase `Exception` y, por lo tanto, es también la superclase de todas las excepciones. Sólo pueden usarse objetos `Throwable` con el mecanismo para manejar excepciones.
- La clase `Throwable` tiene dos subclases: `Exception` y `Error`.
- La clase `Exception` y sus subclases representan situaciones excepcionales que podrían ocurrir en un programa de Java y ser atrapados por la aplicación.
- La clase `Error` y todas sus subclases representan situaciones excepcionales que podrían ocurrir en el sistema en tiempo de ejecución de Java. Los errores tipo `Error` ocurren con poca frecuencia y, por lo general, no deben ser atrapados por una aplicación.
- Java clasifica a las excepciones en dos categorías: verificadas y no verificadas.
- A diferencia de las excepciones verificadas, el compilador de Java no verifica el código para determinar si una excepción no verificada se atrapa o se declara. Por lo general, las excepciones no verificadas se pueden evitar mediante una codificación apropiada.
- El tipo de una excepción determina si ésta es verificada o no verificada. Todos los tipos de excepciones que son subclases directas o indirectas de la clase `RuntimeException` son excepciones no verificadas. Todos los tipos de excepciones que heredan de la clase `Exception` pero no de `RuntimeException` son verificadas.
- Varias clases de excepciones pueden derivarse de una superclase común. Si se escribe un bloque `catch` para atrapar los objetos de excepción de un tipo de la superclase, también puede atrapar a todos los objetos de las subclases de esa clase. Esto permite el procesamiento polimórfico de las excepciones relacionadas.

Sección 13.7 Bloque `finally`

- Los programas que obtienen ciertos tipos de recursos deben devolverlos al sistema de manera explícita, para evitar las denominadas fugas de recursos. Por lo general, el código para liberar recursos se coloca en un bloque `finally`.
- El bloque `finally` es opcional. Si está presente, se coloca después del último bloque `catch`.
- Java garantiza que si se proporciona un bloque `finally`, se ejecutará sin importar que se lance o no una excepción en el bloque `try` correspondiente, o en uno de sus correspondientes bloques `catch`. Java también garantiza que un bloque `finally` se ejecutará si un bloque `try` sale mediante el uso de una instrucción `return`, `break` o `continue`.
- Si una excepción que ocurre en el bloque `try` no se puede atrapar mediante uno de los manejadores `catch` asociados a ese bloque `try`, el programa ignora el resto del bloque `try` y el control pasa al bloque `finally`, que libera el recurso. Después, el programa pasa al siguiente bloque `try` exterior; por lo general, en el método que hace la llamada.
- Si un bloque `catch` lanza una excepción, de todas formas se ejecuta el bloque `finally`. Después, la excepción se pasa al siguiente bloque `try` exterior; por lo general, en el método que hizo la llamada.
- Los programadores pueden lanzar excepciones mediante el uso de la instrucción `throw`.
- Una instrucción `throw` especifica un objeto a lanzar. El operando de una instrucción `throw` puede ser de cualquier clase que se derive de `Throwable`.

Sección 13.8 Limpieza de la pila

- Las excepciones se vuelven a lanzar cuando un bloque `catch`, al momento de recibir una excepción, decide que no puede procesarla, o que sólo puede procesarla en forma parcial. Al volver a lanzar una excepción se difiere el manejo de excepciones (o tal vez una parte de éste) a otro bloque `catch`.
- Cuando se vuelve a lanzar una excepción, el siguiente bloque `try` circundante detecta la excepción que se volvió a lanzar, y los bloques `catch` de ese bloque `try` tratan de manejarla.
- Cuando se lanza una excepción, pero no se atrapa en un alcance específico, se limpia la pila de llamadas a métodos y se hace un intento por atrapar la excepción en la siguiente instrucción `try` exterior. A este proceso se le conoce como limpieza de la pila.

Sección 13.9 `printStackTrace`, `getStackTrace` *y* `getMessage`

- La clase `Throwable` ofrece un método `printStackTrace`, que imprime la pila de llamadas a métodos. A menudo, esto es útil en la prueba y depuración.
- La clase `Throwable` también proporciona un método `getStackTrace`, que obtiene información de rastreo de la pila, que `printStackTrace` imprime.
- El método `getMessage` de la clase `Throwable` devuelve la cadena descriptiva almacenada en una excepción.
- El método `getStackTrace` obtiene la información de rastreo de la pila como un arreglo de objetos `StackTraceElement`. Cada objeto `StackTraceElement` representa una llamada a un método en la pila de llamadas a métodos.
- Los métodos `getClassName`, `getFileName`, `getLineNumber` y `getMethodName` de la clase `StackTraceElement` obtienen el nombre de la clase, el nombre de archivo, el número de línea y el nombre del método, respectivamente.

Sección 13.10 Excepciones encadenadas

- Las excepciones encadenadas permiten que un objeto de excepción mantenga la información de rastreo de la pila completa, incluyendo la información acerca de las excepciones anteriores que provocaron la excepción actual.

Sección 13.11 Declaración de nuevos tipos de excepciones

- Una nueva clase de excepción debe extender a una clase de excepción existente, para asegurar que la clase pueda usarse con el mecanismo de manejo de excepciones.

Sección 13.12 Precondiciones y poscondiciones

- La precondición de un método es una condición que debe ser verdadera al momento de invocar el método.
- La poscondición de un método es una condición que es verdadera una vez que regresa el método con éxito.
- Al diseñar sus propios métodos, debe establecer las precondiciones y poscondiciones en un comentario antes de la declaración del método.

Sección 13.13 Aserciones

- Dentro de una aplicación, los programadores pueden establecer condiciones que asuman como verdaderas en un punto específico. Estas condiciones, conocidas como aserciones, ayudan a asegurar la validez de un programa al atrapar errores potenciales e identificar posibles errores lógicos.
- Java incluye dos versiones de una instrucción `assert` para validar las aserciones mediante la programación.
- Para habilitar las aserciones en tiempo de ejecución, use el modificador `–ea` al ejecutar el comando `java`.

Terminología

`ArithmeticException`, clase
aserción
`assert`, instrucción
atrapar una excepción
bloque `try` circundante
`catch`, bloque
`catch`, cláusula
error sincrónico
`Error`, clase
evento asíncrono
Excepción
excepción encadenada
excepción no atrapada
excepción verificada
excepciones no verificadas
`Exception`, clase
falla en el constructor
`finally`, bloque
`finally`, cláusula
flujo de error estándar
flujo de salida estándar
fuga de recursos
`getClassName`, método de la clase `StackTraceElement`
`getFileName`, método de la clase `StackTraceElement`
`getLineNumber`, método de la clase `StackTraceElement`
`getMessage`, método de la clase `Throwable`
`getMethodName`, método de la clase `StackTraceElement`
`getStackTrace`, método de la clase `Throwable`
`InputMismatchException`, clase
lanzar una excepción
liberar un recurso
limpieza de la pila
manejador de excepciones
manejo de excepciones
modelo de reanudación del manejo de excepciones
modelo de terminación del manejo de excepciones
parámetro de excepción
poscondición
precondición
`printStackTrace`, método de la clase `Throwable`
programa tolerante a fallas
punto de lanzamiento
rastreo de la pila
requerimiento de atrapar o declarar
`RuntimeException`, clase
`StackTraceElement`, clase
`System.err`, flujo
`throw`, instrucción
`throw`, palabra clave
`Throwable`, clase
`throws`, cláusula
`try`, bloque
`try`, instrucción
`try...catch...finally`, mecanismo para manejar excepciones
volver a lanzar una excepción

Ejercicios de autoevaluación

13.1 Enliste cinco ejemplos comunes de excepciones.

13.2 Dé varias razones por las cuales no deban utilizarse las técnicas de manejo de excepciones para el control convencional de los programas.

13.3 ¿Por qué son las excepciones particularmente apropiadas para tratar con los errores producidos por los métodos de las clases en la API de Java?

13.4 ¿Qué es una "fuga de recursos"?

13.5 Si no se lanzan excepciones en un bloque `try`, ¿hacia dónde procede el control cuando el bloque `try` completa su ejecución?

13.6 Mencione una ventaja clave del uso de `catch(Exception` *nombreExcepción*`)`.

13.7 ¿Debe una aplicación convencional atrapar los objetos `Error`? Explique.

13.8 ¿Qué ocurre si ningún manejador `catch` coincide con el tipo de un objeto lanzado?

13.9 ¿Qué ocurre si varios bloques `catch` coinciden con el tipo del objeto lanzado?

13.10 ¿Por qué debería un programador especificar un tipo de superclase como el tipo en un bloque `catch`?

13.11 ¿Cuál es la razón clave de utilizar bloques `finally`?

13.12 ¿Qué ocurre cuando un bloque `catch` lanza una excepción `Exception`?

13.13 ¿Qué hace la instrucción `throw` *referenciaExcepción*?

13.14 ¿Qué ocurre a una referencia local en un bloque `try`, cuando ese bloque lanza una excepción `Exception`?

Respuestas a los ejercicios de autoevaluación

13.1 Agotamiento de memoria, índice de arreglo fuera de límites, desbordamiento aritmético, división entre cero, parámetros inválidos de método.

13.2 a) El manejo de excepciones está diseñado para manejar las situaciones que ocurren con poca frecuencia y que a menudo provocan la terminación del programa, no situaciones que surjan todo el tiempo. b) Por lo general, el flujo de control con estructuras de control convencionales es más claro y eficiente que con las excepciones. c) Las excepciones "adicionales" pueden interponerse en el camino de las excepciones de tipos de errores genuinos. Es más difícil para el programador llevar el registro de un número más extenso de casos de excepciones.

13.3 Es muy poco probable que los métodos de clases en la API de Java puedan realizar un procesamiento de errores que cumpla con las necesidades únicas de todos los usuarios.

13.4 Una "fuga de recursos" ocurre cuando un programa en ejecución no libera apropiadamente un recurso cuando éste ya no es necesario.

13.5 Los bloques `catch` para esa instrucción `try` se ignoran y el programa reanuda su ejecución después del último bloque `catch`. Si hay bloque `finally`, se ejecuta primero y luego el programa reanuda su ejecución después del bloque `finally`.

13.6 La forma `catch(Exception` *nombreExcepción*`)` atrapa cualquier tipo de excepción lanzada en un bloque `try`. Una ventaja es que ninguna excepción `Exception` lanzada puede escabullirse sin ser atrapada. El programador puede entonces decidir si manejará la excepción o si posiblemente vuelva a lanzarla.

13.7 Las excepciones `Error` son generalmente problemas graves con el sistema de Java subyacente; en la mayoría de los programas no es conveniente atrapar excepciones `Error`, ya que el programa no podrá recuperarse de dichos problemas.

13.8 Esto hace que la búsqueda de una coincidencia continúe en la siguiente instrucción `try` circundante. Si hay un bloque `finally`, éste se ejecutará antes de que la excepción pase a la siguiente instrucción `try` circundante. Si no hay instrucciones `try` circundantes para las cuales haya bloques `catch` que coincidan, y la excepción es *verificada*, se produce un error de compilación. Si no hay instrucciones `try` circundantes para las cuales haya bloques `catch` que coincidan y la excepción es *no verificada*, se imprime un rastreo de la pila y el subproceso actual termina antes de tiempo.

13.9 Se ejecuta el primer bloque `catch` que coincida después del bloque `try`.

13.10 Esto permite a un programa atrapar tipos relacionados de excepciones, y procesarlos en una manera uniforme. Sin embargo, a menudo es conveniente procesar los tipos de subclases en forma individual, para un manejo de excepciones más preciso.

13.11 La cláusula `finally` es el medio preferido para liberar recursos y evitar las fugas de éstos.

13.12 Primero, el control pasa al bloque `finally`, si existe uno. Después, la excepción se procesará mediante un bloque `catch` (si existe uno) asociado con un bloque `try` circundante (si existe uno).

13.13 Vuelve a lanzar la excepción para que la procese un manejador de excepciones de un bloque `try` circundante, una vez que se ejecuta el bloque `finally` de la instrucción `try` actual.

13.14 La referencia queda fuera de alcance, y la cuenta de referencias para el objeto se decrementa. Si la cuenta de referencias se vuelve cero, el objeto se marca para la recolección de basura.

Ejercicios

13.15 Enliste las diversas condiciones excepcionales que han ocurrido en programas, a lo largo de este texto. Enliste todas las condiciones excepcionales adicionales que pueda. Para cada una de ellas, describa brevemente la manera en que un programa manejaría la excepción, utilizando las técnicas de manejo de excepciones que se describen en este capítulo. Algunas excepciones típicas son la división entre cero, el desbordamiento aritmético y el índice de arreglo fuera de límites.

13.16 Hasta este capítulo, hemos visto que tratar con los errores detectados por los constructores es algo difícil. Explique por qué el manejo de excepciones es un medio efectivo para tratar con las fallas en los constructores.

13.17 *(Atrapar excepciones con las superclases)* Use la herencia para crear una superclase de excepción (llamada `ExcepcionA`) y las subclases de excepción `ExcepcionB` y `ExcepcionC`, en donde `ExcepcionB` hereda de `ExcepcionA` y `ExcepcionC` hereda de `ExcepcionB`. Escriba un programa para demostrar que el bloque `catch` para el tipo `ExcepcionA` atrapa excepciones de los tipos `ExcepcionB` y `ExcepcionC`.

13.18 *(Atrapar excepciones mediante el uso de la clase `Exception`)* Escriba un programa que demuestre cómo se atrapan las diversas excepciones mediante catch con

```
catch ( Exception excepcion )
```

Esta vez, defina las clases `ExcepcionA` (que hereda de la clase `Exception`) y `ExcepcionB` (que hereda de la clase `ExcepcionA`). En su programa, cree bloques `try` que lancen excepciones de los tipos `ExcepcionA`, `ExcepcionB`, `NullPointerException` e `IOException`. Todas las excepciones deberán atraparse con bloques `catch` que especifiquen el tipo `Exception`.

13.19 *(Orden de los bloques `catch`)* Escriba un programa que demuestre que el orden de los bloques `catch` es importante. Si trata de atrapar un tipo de excepción de superclase antes de un tipo de subclase, el compilador debe generar errores.

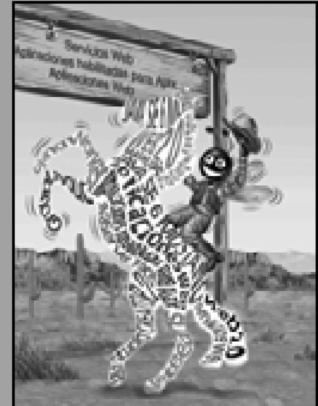
13.20 *(Falla del constructor)* Escribe un programa que muestre cómo un constructor pasa información sobre la falla del constructor a un manejador de excepciones. Defina la clase `unaExcepcion`, que lanza una excepción `Exception` en el constructor. Su programa deberá tratar de crear un objeto de tipo `UnaExcepcion` y atrapar la excepción que se lance desde el constructor.

13.21 *(Volver a lanzar expresiones)* Escriba un programa que ilustre cómo volver a lanzar una excepción. Defina los métodos `unMetodo` y `unMetodo2`. El método `unMetodo2` debe lanzar al principio una excepción. El método `unMetodo` debe llamar a `unMetodo2`, atrapar la excepción y volver a lanzarla. Llame a `unMetodo` desde el método `main`, y atrape la excepción que se volvió a lanzar. Imprima el rastreo de la pila de esta excepción.

13.22 *(Atrapar excepciones mediante el uso de alcances exteriores)* Escriba un programa que muestre que un método con su propio bloque `try` no tiene que atrapar todos los posibles errores que se generen dentro del `try`. Algunas excepciones pueden pasarse hacia otros alcances, en donde se manejan.

14

Archivos y flujos



Sólo puedo suponer que un documento “No archivar” se archiva en un archivo “No archivar”.

—Senador Frank Church
Audiencia del subcomité de inteligencia del Senado, 1975

La conciencia… no aparece a sí misma cortada en pequeños pedazos…
Un “río” o un “flujo” son las metáforas por las cuales se describe con más naturalidad.

—William James

Leí una parte en su totalidad.

—Samuel Goldwyn

Una gran memoria no hace a un filósofo; cualquier cosa que sea más que un diccionario se le puede llamar gramática.

—John Henry, Cardenal Newman

OBJETIVOS

En este capítulo aprenderá a:

- Crear, leer, escribir y actualizar archivos.
- Utilizar la clase `File` para obtener información acerca de los archivos y directorios.
- Comprender la jerarquía de clases de flujos de entrada/salida en Java.
- Conocer las diferencias entre los archivos de texto y los archivos binarios.
- Comprender el procesamiento de archivos de acceso secuencial.
- Utilizar las clases `Scanner` y `Formatter` para procesar archivos de texto.
- Utilizar las clases `FileInputStream` y `FileOutputStream`.
- Utilizar un cuadro de diálogo `JFileChooser`.
- Utilizar las clases `ObjectInputStream` y `ObjectOutputStream`.

Plan general

14.1 Introducción

El almacenamiento de datos en variables y arreglos es temporal; los datos se pierden cuando una variable local queda fuera de alcance, o cuando el programa termina. Las computadoras utilizan archivos para la retención a largo plazo de grandes cantidades de datos, incluso hasta después de que terminan los programas que crean esos datos. Usted utiliza archivos a diario, para tareas como escribir un ensayo o crear una hoja de cálculo. Nos referimos a los datos que se mantienen en archivos como **datos persistentes**, ya que existen más allá de la duración de la ejecución del programa. Las computadoras almacenan archivos en **dispositivos de almacenamiento secundario** como discos duros, discos ópticos y cintas magnéticas. En este capítulo explicaremos cómo los programas en Java crean, actualizan y procesan archivos.

El procesamiento de archivos es una de las herramientas más importantes que debe tener un lenguaje para soportar las aplicaciones comerciales, que generalmente procesan cantidades masivas de datos persistentes. En este capítulo hablaremos sobre las poderosas características de procesamiento de archivos y flujos de entrada/salida de Java. El término "flujo" se refiere a los datos ordenados que se leen de (o se escriben en) un archivo. En la sección 14.3 hablaremos con más detalle sobre los flujos. El procesamiento de archivos es un subconjunto de las herramientas para procesar flujos de Java, las cuales permiten a un programa leer y escribir datos en memoria, en archivos y a través de conexiones de red. Tenemos dos metas en este capítulo: introducir los conceptos acerca del procesamiento de archivos (para que el lector se sienta más familiarizado con el uso de los archivos en la programación) y proporcionar al lector las suficientes herramientas de procesamiento de archivos como para soportar las características de red que se presentan en el capítulo 24, Redes. Java cuenta con importantes herramientas de procesamiento de flujos; más de lo que podemos cubrir en un capítulo. Aquí hablaremos sobre dos formas de procesamiento de archivos: el procesamiento de archivos de texto y la serialización de objetos.

Empezaremos hablando sobre la jerarquía de los datos contenidos en archivos. Después veremos la arquitectura de Java para manejar archivos mediante programación; hablaremos sobre varias clases del paquete `java.io.` Luego explicaremos que los datos pueden almacenarse en dos tipos distintos de archivos (de texto y binarios) y cubriremos las diferencias entre ellos. Demostraremos cómo obtener información acerca de un archivo o directorio mediante el uso de la clase `File`, y después dedicaremos varias secciones a los distintos mecanismos para escribir datos en (y leer datos de) archivos. Primero demostraremos cómo crear y manipular archivos de texto de acceso secuencial. Al trabajar con archivos de texto, el lector puede empezar a manipular archivos con rapidez y facilidad. Sin embargo, como veremos más adelante, es difícil leer datos de los archivos de texto y devolverlos al formato de los objetos. Por fortuna, muchos lenguajes orientados a objetos (incluyendo Java) ofrecen distintas formas de escribir objetos en (y leer objetos de) archivos (lo que se conoce como serialización y deserialización de

objetos). Para demostrar esto, recreamos algunos de los programas de acceso secuencial que utilizaban archivos de texto, esta vez almacenando objetos en archivos binarios.

14.2 Jerarquía de datos

Básicamente, una computadora procesa todos los elementos de datos como combinaciones de ceros y unos, ya que para los ingenieros es sencillo y económico construir dispositivos electrónicos que puedan suponer dos estados estables: uno representa `0` y el otro, `1`. Es increíble que las impresionantes funciones realizadas por las computadoras impliquen solamente las manipulaciones más fundamentales de `0`s y `1`s.

El elemento más pequeño de datos en una computadora puede asumir el valor `0` o `1`. Dicho elemento de datos se conoce como **bit** (abreviatura de "dígito binario"; un dígito que puede suponer uno de dos valores). Los circuitos de computadora realizan varias manipulaciones simples de bits, como examinar o establecer el valor de un bit, o invertir su valor (de `1` a `0` o de `0` a `1`).

Es muy difícil para los programadores trabajar con datos en el formato de bits de bajo nivel. En vez de ello, los programadores prefieren trabajar con datos en formatos como **dígitos decimales** (0-9), **letras** (A-Z y a-z) y **símbolos especiales** (por ejemplo, $, @, %, &, *, (,), —, +, ", :, ? y /). Los dígitos, letras y símbolos especiales se conocen como **caracteres**. El **conjunto de caracteres** de la computadora es el conjunto de todos los caracteres utilizados para escribir programas y representar elementos de datos. Las computadoras pueden procesar solamente `1`s y `0`s, por lo que un conjunto de caracteres representa a todos los caracteres como un patrón de `1`s y `0`s. Los caracteres en Java son caracteres **Unicode**, compuestos de dos **bytes**. Cada byte está compuesto de ocho bits. Java contiene un tipo de datos, `byte`, que pueden usarse para representar datos tipo byte. El conjunto de caracteres Unicode contiene caracteres para muchos de los lenguajes utilizados en todo el mundo. En el apéndice I podrá obtener más información acerca de este conjunto de caracteres. En el apéndice B, Conjunto de caracteres ASCII, podrá obtener más información acerca del conjunto de caracteres **ASCII (Código Estándar Estadounidense para el Intercambio de Información)**, un subconjunto del conjunto de caracteres Unicode que representa letras mayúsculas y minúsculas, dígitos y varios caracteres especiales comunes.

Así como los caracteres están compuestos de bits, los **campos** están compuestos de caracteres o bytes. Un campo es un grupo de caracteres o bytes que transmiten cierto significado. Por ejemplo, un campo que consiste de letras mayúsculas y minúsculas puede utilizarse para representar el nombre de una persona.

Los elementos de datos que son procesados por las computadoras forman una **jerarquía de datos**, la cual se hace más grande y compleja en estructura, a medida que progresamos de bits a caracteres, de caracteres a campos, etcétera.

Generalmente, varios campos forman un **registro** (que se implementa como `class` en Java). Por ejemplo, en un sistema de nóminas el registro para un empleado podría estar compuesto de los siguientes campos (los posibles tipos para estos campos se muestran entre paréntesis):

- Número de identificación del empleado (`int`).
- Nombre (`String`).
- Dirección (`String`).
- Sueldo por hora (`double`).
- Número de exepciones reclamadas (`int`).
- Ingresos desde inicio de año a la fecha (`int` o `double`).
- Monto de impuestos retenidos (`int` o `double`).

Por lo tanto, un registro es un grupo de campos relacionados. En el ejemplo anterior, cada uno de los campos pertenece al mismo empleado. Desde luego que una compañía específica podría tener muchos empleados y, por ende, tendría un registro de nómina para cada empleado. Un **archivo** es un grupo de registros relacionados. [*Nota:* dicho en forma más general, un archivo contiene datos arbitrarios en formatos arbitrarios. En algunos sistemas operativos, un archivo se ve simplemente como una colección de bytes; cualquier organización de los bytes en un archivo (como organizar los datos en registros) es una vista creada por el programador de aplicaciones]. El archivo de nómina de una compañía generalmente contiene un registro para cada empleado. Por ejemplo, un archivo de nómina para una pequeña compañía podría contener sólo 22 registros, mientras que un archivo de nómina para una compañía grande podría contener 100,000 registros. Es común para una compañía tener muchos archivos, algunos de ellos conteniendo miles de millones, o incluso billones de caracteres de información. En la figura 14.1 se muestra una parte de la jerarquía de datos.

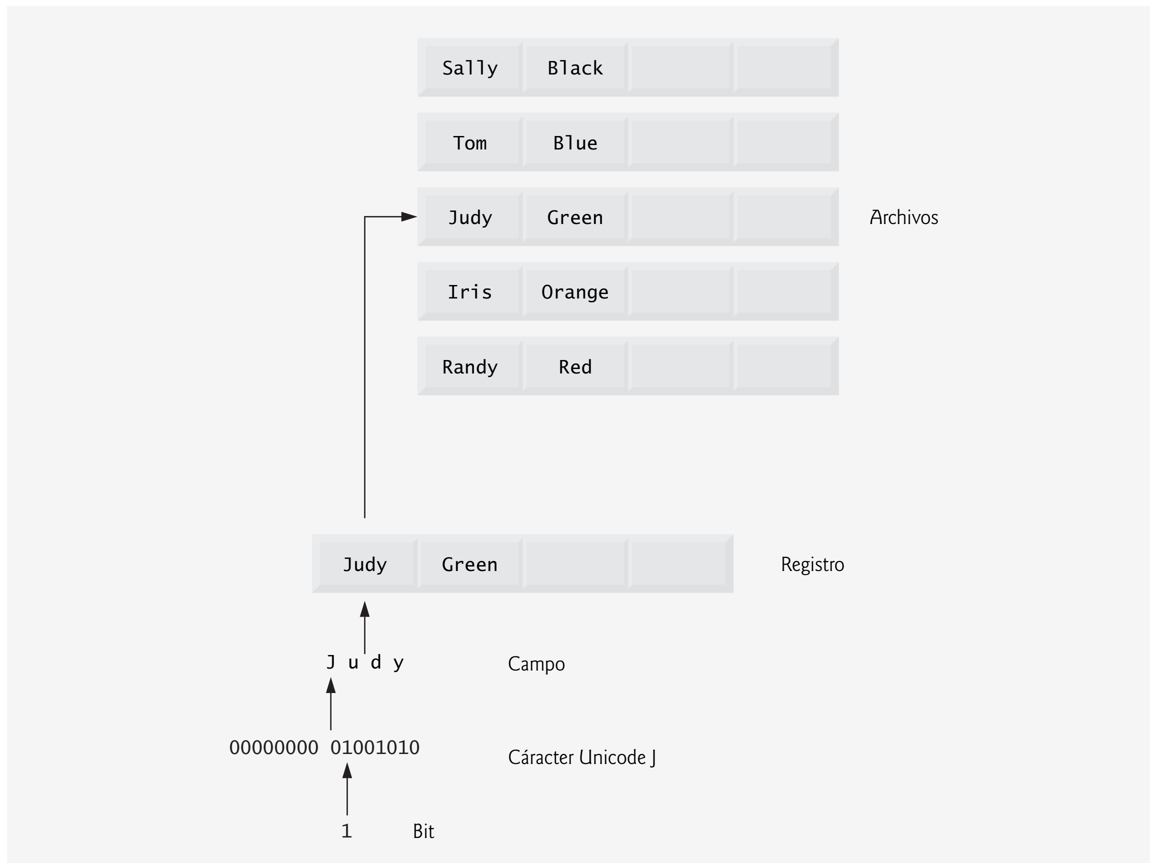


Figura 14.1 | Jerarquía de datos.

Para facilitar la recuperación de registros específicos de un archivo, debe seleccionarse cuando menos un campo en cada registro como **clave de registro**. Una clave de registro sirve para identificar que un registro pertenece a una persona o entidad específica, y es única en cada registro. Este campo generalmente se utiliza para buscar y ordenar registros. En el registro de nómina que describimos anteriormente, por lo general, se elegiría el número de identificación de empleado como clave de registro.

Existen muchas formas de organizar los registros en un archivo. La organización más común se conoce como **archivo secuencial**, en el cual los registros se almacenan en orden, en base al campo que es la clave de registro. En un archivo de nómina, los registros se colocan en orden, en base al número de identificación de empleado.

La mayoría de las empresas almacena datos en muchos archivos distintos. Por ejemplo, las compañías podrían tener archivos de nómina, de cuentas por cobrar (listas del dinero que deben los clientes), de cuentas por pagar (listas del dinero que se debe a los proveedores), archivos de inventarios (listas de información acerca de los artículos que maneja la empresa) y muchos otros tipos de archivos. A menudo, a un grupo de archivos relacionados se le conoce como **base de datos**. A una colección de programas diseñada para crear y administrar bases de datos se le conoce como **sistema de administración de bases de datos (DBMS)**. Hablaremos sobre las bases de datos en el capítulo 25, Acceso a bases de datos con JDBC.

14.3 Archivos y flujos

Java considera a cada archivo como un **flujo** secuencial de bytes (figura 14.2). Cada sistema operativo proporciona un mecanismo para determinar el fin de un archivo, como el **marcador de fin de archivo** o la cuenta de bytes totales en el archivo que se registra en una estructura de datos administrativa, mantenida por el sistema. Un programa de Java que procesa un flujo de bytes simplemente recibe una indicación del sistema operativo cuando

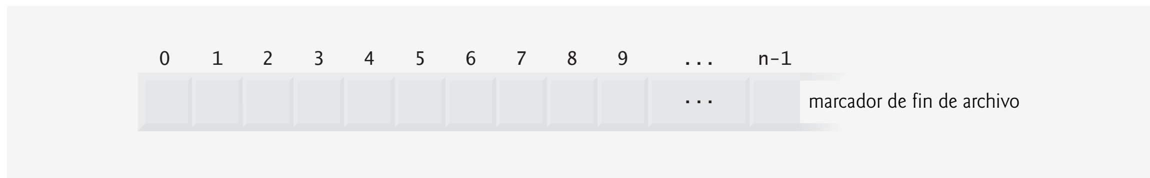


Figura 14.2 | La manera en que Java ve a un archivo de *n* bytes.

el programa llega al fin del flujo; el programa no necesita saber cómo representa la plataforma subyacente a los archivos o flujos. En algunos casos, la indicación de fin de archivo ocurre como una excepción. En otros casos, la indicación es un valor de retorno de un método invocado en un objeto procesador de flujos.

Los flujos de archivos se pueden utilizar para la entrada y salida de datos, ya sea como caracteres o bytes. Los flujos que reciben y envían bytes a archivos se conocen como **flujos basados en bytes**, y almacenan datos en su formato binario. Los flujos que reciben y envían caracteres de/a los archivos se conocen como **flujos basados en caracteres**, y almacenan datos como una secuencia de caracteres. Por ejemplo, si se almacenara el valor `5` usando un flujo basado en bytes, sería en el formato binario del valor numérico `5`, o `101`. Si se almacenara el valor `5` usando un flujo basado en caracteres, sería en el formato binario del carácter `5`, o `00000000 00110101` (ésta es la representación binaria para el valor numérico `53`, el cual indica el carácter `5` en el conjunto de caracteres Unicode). La diferencia entre el valor numérico `5` y el carácter `5` es que el valor numérico se puede utilizar como un entero en los cálculos, mientras que el carácter `5` es simplemente un carácter que puede utilizarse en una cadena de texto, como en `"Sarah Miller tiene 15 años de edad"`. Los archivos que se crean usando flujos basados en bytes se conocen como **archivos binarios**, mientras que los archivos que se crean usando flujos basados en caracteres se conocen como **archivos de texto**. Los archivos de texto se pueden leer con editores de texto, mientras que los archivos binarios se leen mediante un programa que convierte los datos en un formato que pueden leer los humanos.

Un programa de Java **abre** un archivo creando un objeto y asociándole un flujo de bytes o de caracteres. En breve hablaremos sobre las clases que se utilizan para crear esos objetos. Java también puede asociar flujos de bytes con distintos dispositivos. De hecho, Java crea tres objetos flujo que se asocian con dispositivos cuando un programa de Java empieza a ejecutarse: `System.in`, `System.out` y `System.err` El objeto `System.in` (el objeto flujo de entrada estándar) generalmente permite a un programa recibir bytes desde el teclado; el objeto `System.out` (el objeto flujo estándar de salida) generalmente permite a un programa mostrar datos en la pantalla; y el objeto `System.err` (el objeto flujo estándar de error) generalmente permite a un programa mostrar mensajes de error en la pantalla. Cada uno de estos flujos puede **redirigirse**. Para `System.in`, esta capacidad permite al programa leer bytes desde un origen distinto. Para `System.out` y `System.err`, esta capacidad permite que la salida se envíe a una ubicación distinta, como un archivo en disco. La clase `System` proporciona los métodos `setIn`, `setOut` y `setErr` para redirigir los flujos estándar de entrada, salida y error, respectivamente.

Los programas de Java realizan el procesamiento de archivos utilizando clases del paquete `java.io`. Este paquete incluye definiciones para las clases de flujo como `FileInputStream` (para la entrada basada en bytes desde un archivo), `FileOutputStream` (para la salida basada en bytes hacia un archivo), `FileReader` (para la entrada basada en caracteres desde un archivo) y `FileWriter` (para la salida basada en caracteres hacia un archivo). Los archivos se abren creando objetos de estas clases de flujos, que heredan de las clases `InputStream`, `OutputStream`, `Reader` y `Writer`, respectivamente (más adelante en este capítulo hablaremos sobre estas clases). Por lo tanto, los métodos de estas clases de flujos pueden aplicarse a los flujos de archivos también.

Java contiene clases que permiten al programador realizar operaciones de entrada y salida con objetos o variables de tipos de datos primitivos. Los datos se siguen almacenando como bytes o caracteres tras bambalinas, lo cual permite al programador leer o escribir datos en forma de enteros, cadenas u otros tipos de datos, sin tener que preocuparse por los detalles acerca de convertir dichos valores al formato de bytes. Para realizar dichas operaciones de entrada y salida, pueden usarse objetos de las clases `ObjectInputStream`, y `ObjectOutputStream` junto con las clases de flujos de archivos basadas en bytes `FileInputStream` y `FileOutputStream` (en breve hablaremos con más detalle sobre estas clases). La jerarquía completa de clases en el paquete `java.io` puede consultarse en la documentación en línea, en la página:

```
java.sun.com/javase/6/docs/api/java/io/package-tree.html
```

En la jerarquía, cada nivel de sangría indica que la clase con sangría extiende a la clase encima de ella. Por ejemplo, la clase `InputStream` es una subclase de `Object`. Haga clic en el nombre de una clase en la jerarquía para ver los detalles de esa clase.

Como puede ver en la jerarquía, Java ofrece muchas clases para realizar operaciones de entrada/salida. En este capítulo usaremos varias de estas clases para implementar programas de procesamiento de archivos, que crean y manipulan archivos de acceso secuencial. También incluiremos un ejemplo detallado acerca de la clase `File`, que es útil para obtener información sobre archivos y directorios. En el capítulo 24, Redes, utilizaremos las clases de flujos en forma extensa, para implementar aplicaciones de red. En la sección 14.7 hablaremos brevemente sobre varias otras clases del paquete `java.io` que no usaremos en este capítulo.

Además de las clases en este paquete, las operaciones de entrada y salida basadas en caracteres se pueden llevar a cabo con las clases `Scanner` y `Formatter`. La clase `Scanner` se utiliza en forma extensa para recibir datos del teclado. Como veremos, esta clase también puede leer datos de un archivo. La clase `Formatter` permite mostrar datos con formato en la pantalla, o enviarlos a un archivo, en forma similar a `System.out.printf`. En el capítulo 29, Salida con formato, se presentan los detalles acerca de la salida con formato mediante `System.out.printf`. Todas estas características se pueden utilizar también para dar formato a los archivos de texto.

14.4 La clase `File`

En esta sección presentamos la clase `File`, que es especialmente útil para recuperar información acerca de un archivo o directorio de un disco. Los objetos de la clase `File` no abren archivos ni proporcionan herramientas para procesarlos. No obstante, los objetos `File` se utilizan frecuentemente con objetos de otras clases de `java.io` para especificar los archivos o directorios que van a manipularse.

Creación de objetos `File`

La clase `File` proporciona cuatro constructores. El constructor:

```
public File( String nombre )
```

especifica el nombre de un archivo o directorio que se asociará con el objeto `File`. El `nombre` puede contener **información sobre la ruta**, así como el nombre de un archivo o directorio. La ruta de un archivo o directorio especifica su ubicación en el disco. La ruta incluye algunos o todos los directorios que conducen a ese archivo o directorio. Una **ruta absoluta** contiene todos los directorios, empezando con el **directorio raíz**, que conducen a un archivo o directorio específico. Cada archivo o directorio en un disco duro específico tiene el mismo directorio raíz en su ruta. Una **ruta relativa** normalmente empieza desde el directorio en el que la aplicación empezó a ejecutarse y es, por lo tanto, una ruta "relativa" al directorio actual.

El constructor:

```
public File( String rutaAlNombre, String nombre )
```

usa el argumento `rutaAlNombre` (una ruta absoluta o relativa) para localizar el archivo o directorio especificado por nombre.

El constructor:

```
public File( File directorio, String nombre )
```

usa un objeto `File` existente llamado `directorio` (una ruta absoluta o relativa) para localizar el archivo o directorio especificado por `nombre`. En la figura 14.3 se enlistan algunos métodos comunes de `File`. La lista completa puede verse en `java.sun.com/javase/6/docs/api/java/io/File.html`.

El constructor:

```
public File( URI uri )
```

usa el objeto `URI` dado para localizar el archivo. Un **Identificador uniforme de recursos (URI)** es una forma más general de un **Localizador uniforme de recursos (URL)**, el cual se utiliza comúnmente para localizar sitios Web. Por ejemplo, `http://www.deitel.com/` es el URL para el sitio Web de Deitel & Associates. Los URIs para localizar archivos varían entre los distintos sistemas operativos. En plataformas Windows, el URI:

```
file:/C:/datos.txt
```

identifica al archivo `datos.txt`, almacenado en el directorio raíz de la unidad C:. En plataformas UNIX/Linux, el URI

```
file:/home/estudiante/datos.txt
```

identifica el archivo `datos.txt` almacenado en el directorio `home` del usuario `estudiante`.

Tip para prevenir errores 14.1

Use el método `isFile` de `File` para determinar si un objeto `File` representa a un archivo (y no a un directorio) antes de tratar de abrir el archivo.

Método	Descripción
`boolean canRead()`	Devuelve `true` si la aplicación actual puede leer un archivo; `false` en caso contrario.
`boolean canWrite()`	Devuelve `true` si la aplicación actual puede escribir en un archivo; `false` en caso contrario.
`boolean exists()`	Devuelve `true` si el nombre especificado como argumento para el constructor de `File` es un archivo o directorio en la ruta especificada; `false` en caso contrario.
`boolean isFile()`	Devuelve `true` si el nombre especificado como argumento para el constructor de `File` es un archivo; `false` en caso contrario.
`boolean isDirectory()`	Devuelve `true` si el nombre especificado como argumento para el constructor de `File` es un directorio; `false` en caso contrario.
`boolean isAbsolute()`	Devuelve `true` si los argumentos especificados para el constructor de `File` indican una ruta absoluta a un archivo o directorio; `false` en caso contrario.
`String getAbsolutePath()`	Devuelve una cadena con la ruta absoluta del archivo o directorio.
`String getName()`	Devuelve una cadena con el nombre del archivo o directorio.
`String getPath()`	Devuelve una cadena con la ruta del archivo o directorio.
`String getParent()`	Devuelve una cadena con el directorio padre del archivo o directorio (es decir, el directorio en el que puede encontrarse ese archivo o directorio).
`long length()`	Devuelve la longitud del archivo, en bytes. Si el objeto `File` representa a un directorio, se devuelve `0`.
`long lastModified()`	Devuelve una representación dependiente de la plataforma de la hora en la que se hizo la última modificación en el archivo o directorio. El valor devuelto es útil sólo para compararlo con otros valores devueltos por este método.
`String[] list()`	Devuelve un arreglo de cadenas, las cuales representan el contenido de un directorio. Devuelve `null` si el objeto `File` no representa a un directorio.

Figura 14.3 | Métodos de `File`.

Demostración de la clase `File`

Las figuras 14.4 y 14.5 demuestran el uso de la clase `File`. La aplicación pide al usuario que introduzca el nombre de un archivo o directorio, y después imprime información en pantalla acerca del nombre de archivo o directorio introducido.

El programa empieza pidiendo al usuario un archivo o directorio (línea 12 de la figura 14.5). En la línea 13 se introduce el nombre del archivo o directorio y se pasa al método `analizarRuta` (líneas 8 a 41 de la figura 14.4). El método crea un nuevo objeto `File` (línea 11) y asigna su referencia a `nombre`. En la línea 13 se invoca el método

`exists` de `File` para determinar si el nombre introducido por el usuario existe (ya sea como archivo o directorio) en el disco. Si el nombre introducido por el usuario no existe, el control procede a las líneas 37 a 40 y muestra un mensaje en la pantalla, que contiene el nombre que escribió el usuario, seguido de "`no existe`". En caso contrario, se ejecuta el cuerpo de la instrucción `if` (líneas 13 a 36). El programa imprime el nombre del archivo o directorio (línea 18), seguido de los resultados de probar el objeto `File` con `isFile` (línea 19), `isDirectory` (línea 20) e `isAbsolute` (línea 22). A continuación, el programa muestra los valores devueltos por `lastModified` (línea 24), `length` (línea 24), `getPath` (línea 25), `getAbsolutePath` (línea 26) y `getParent` (línea 26). Si el objeto `File` representa un directorio (línea 28), el programa obtiene una lista del contenido del directorio como un arreglo de objetos `String`, usando el método `list` de `File` (línea 30), y muestra la lista en la pantalla.

El primer resultado de este programa demuestra un objeto `File` asociado con el directorio `jfc` del Kit de Desarrollo de Software de Java 2. El segundo resultado demuestra un objeto `File` asociado con el archivo `readme.txt` del ejemplo de Java 2D que viene con el Kit de Desarrollo de Software de Java 2D. En ambos casos, especificamos una ruta absoluta en nuestra computadora personal.

```
1  // Fig. 14.4: DemostracionFile.java
2  // Demostración de la clase File.
3  import java.io.File;
4
5  public class DemostracionFile
6  {
7     // muestra información acerca del archivo especificado por el usuario
8     public void analizarRuta( String ruta )
9     {
10       // crea un objeto File con base en la entrada del usuario
11       File nombre = new File( ruta );
12
13       if ( nombre.exists() ) // si existe el nombre, muestra información sobre él
14       {
15          // muestra información del archivo (o directorio)
16          System.out.printf(
17             "%s%s\n%s\n%s\n%s\n%s%s\n%s%s\n%s%s\n%s%s\n%s%s",
18             nombre.getName(), " existe",
19             ( nombre.isFile() ? "es un archivo" : "no es un archivo" ),
20             ( nombre.isDirectory() ? "es un directorio" :
21                "no es un directorio" ),
22             ( nombre.isAbsolute() ? "es ruta absoluta" :
23                "no es ruta absoluta" ), "Ultima modificacion: ",
24             nombre.lastModified(), "Tamanio: ", nombre.length(),
25             "Ruta: ", nombre.getPath(), "Ruta absoluta: ",
26             nombre.getAbsolutePath(), "Padre: ", nombre.getParent() );
27
28          if ( nombre.isDirectory() ) // muestra el listado del directorio
29          {
30             String directorio[] = nombre.list();
31             System.out.println( "\n\nContenido del directorio:\n" );
32
33             for ( String nombreDirectorio : directorio )
34                System.out.printf( "%s\n", nombreDirectorio );
35          } // fin de else
36       } // fin de if exterior
37       else // no es archivo o directorio, muestra mensaje de error
38       {
39          System.out.printf( "%s %s", ruta, "no existe." );
40       } // fin de else
41    } // fin del método analizarRuta
42 } // fin de la clase DemostracionFile
```

Figura 14.4 | Uso de la clase `File` para obtener información sobre archivos y directorios.

```java
// Fig. 14.5: PruebaDemostracionFile.java
// Prueba de la clase DemostracionFile.
import java.util.Scanner;

public class PruebaDemostracionFile
{
   public static void main( String args[] )
   {
      Scanner entrada = new Scanner( System.in );
      DemostracionFile aplicacion = new DemostracionFile();

      System.out.print( "Escriba aqui el nombre del archivo o directorio: " );
      aplicacion.analizarRuta( entrada.nextLine() );
   } // fin de main
} // fin de la clase PruebaDemostracionFile
```

```
Escriba aqui el nombre del archivo o directorio: C:\Archivos de programa\Java\jdk1.6.0_01\
demo\jfc
jfc existe
no es un archivo
es un directorio
es ruta absoluta
Ultima modificacion: 1187324492359
Tamanio: 0
Ruta: C:\Archivos de programa\Java\jdk1.6.0_01\demo\jfc
Ruta absoluta: C:\Archivos de programa\Java\jdk1.6.0_01\demo\jfc
Padre: C:\Archivos de programa\Java\jdk1.6.0_01\demo

Contenido del directorio:

CodePointIM
FileChooserDemo
Font2DTest
Java2D
Metalworks
Notepad
SampleTree
Stylepad
SwingApplet
SwingSet2
TableExample
```

```
Escriba aqui el nombre del archivo o directorio: C:\Archivos de programa\Java\jdk1.6.0_01\
demo\jfc\Java2D\readme.txt
readme.txt existe
es un archivo
no es un directorio
es ruta absoluta
Ultima modificacion: 1187324491203
Tamanio: 7518
Ruta: C:\Archivos de programa\Java\jdk1.6.0_01\demo\jfc\Java2D\readme.txt
Ruta absoluta: C:\Archivos de programa\Java\jdk1.6.0_01\demo\jfc\Java2D\readme.txt
Padre: C:\Archivos de programa\Java\jdk1.6.0_01\demo\jfc\Java2D
```

Figura 14.5 | Prueba de la clase `DemostracionFile`.

Un **carácter separador** se utiliza para separar directorios y archivos en la ruta. En un equipo Windows, el carácter separador es la barra diagonal inversa (\). En una estación de trabajo UNIX, el carácter separador es la

barra diagonal (/). Java procesa ambos caracteres en forma idéntica en el nombre de una ruta. Por ejemplo, si deseamos utilizar la ruta

```
c:\Archivos de programa\Java\jdk1.6.0\demo/jfc
```

que emplea uno de cada uno de los caracteres separadores antes mencionados, Java de todas formas procesa la ruta apropiadamente. Al construir cadenas que representen la información de una ruta, use `File.separator` para obtener el carácter separador apropiado del equipo local, en vez de utilizar `/` o `\` de manera explícita. Esta constante devuelve un objeto `String` que consiste de un carácter: el separador apropiado para el sistema.



Error común de programación 14.1

Usar `\` *como separador de directorios en vez de* `\\` *en una literal de cadena es un error lógico. Una sola* `\` *indica que la* `\` *y el siguiente carácter representan una secuencia de escape. Para insertar una* `\` *en una literal de cadena, debe usar* `\\`.

14.5 Archivos de texto de acceso secuencial

En esta sección crearemos y manipularemos archivos de acceso secuencial. Como dijimos antes, éstos son archivos en donde se guardan los registros en orden, en base al campo clave de registro. Primero demostraremos los archivos de acceso secuencial usando archivos de texto, para permitir al lector crear y editar rápidamente archivos que puedan ser leídos por los humanos. En las subsecciones de este capítulo hablaremos sobre crear, escribir datos en, leer datos de y actualizar los archivos de texto de acceso secuencial. También incluiremos un programa de consulta de crédito para obtener datos específicos de un archivo.

14.5.1 Creación de un archivo de texto de acceso secuencial

Java no impone una estructura en un archivo; lo que los conceptos como un registro no existen en los archivos de Java. Por lo tanto, el programador debe estructurar los archivos de manera que cumplan con los requerimientos de sus aplicaciones. En el siguiente ejemplo veremos cómo el programador puede imponer una estructura de registros en un archivo.

El programa de las figuras 14.6 a 14.7 y en la figura 14.9 crea un archivo simple de acceso secuencial, que podría utilizarse en un sistema de cuentas por cobrar para ayudar a administrar el dinero que deben a una compañía los clientes a crédito. Por cada cliente, el programa obtiene un número de cuenta, el nombre del cliente y su saldo (es decir, el monto que el cliente aún debe a la compañía por los bienes y servicios recibidos). Los datos obtenidos para cada cliente constituyen un "registro" para ese cliente. El número de cuenta se utiliza como la clave de registro en esta aplicación; el archivo se creará y mantendrá en orden basado en el número de cuenta. El programa supone que el usuario introduce los registros en orden de número de cuenta. En un sistema comprensivo de cuentas por cobrar (basado en archivos de acceso secuencial), se proporcionaría una herramienta para ordenar datos, de manera que el usuario pudiera introducir los registros en cualquier orden. Después, los registros se ordenarían y se escribirían en el archivo.

La clase `RegistroCuenta` (figura 14.6) encapsula la información de registro del cliente (es decir, número de cuenta, primer nombre, etcétera) utilizada por los ejemplos en este capítulo. La clase `RegistroCuenta` se declara en el paquete `com.deitel.jhtp7.cap14` (línea 3), de forma que se pueda importar en varios ejemplos. La clase `RegistroCuenta` contiene los miembros de datos `private` llamados `cuenta`, `primerNombre`, `apellidoPaterno` y `saldo` (líneas 7 a 10). Esta clase también proporciona métodos públicos *establecer* y *obtener* para acceder a los campos `private`.

Compile la clase `RegistroCuenta` de la siguiente manera:

```
javac -d  c:\ejemplos\cap14 com\deitel\jhtp7\cap14\RegistroCuenta.java
```

Esto coloca a `RegistroCuenta.class` en la estructura de directorios de su paquete, y coloca el paquete en `c:\ejemplos\cap14`. Cuando compile la clase `RegistroCuenta` (o cualquier otra clase que se reutilice en este capítulo), debe colocarla en un directorio común (por ejemplo, `c:\ejemplos\cap14`). Cuando compile o ejecute clases que utilicen a `RegistroCuenta` (por ejemplo, `CrearArchivoTexto` en la figura 14.7), debe especificar el argumento de línea de comandos `-classpath` para `javac` y `java`, como en

```
javac -classpath .;c:\ejemplos\cap14 CrearArchivoTexto.java
java -classpath .;c:\ejemplos\cap14 CrearArchivoTexto
```

```
// Fig. 14.6: RegistroCuenta.java
// Una clase que representa un registro de información
package com.deitel.jhtp7.cap14; // se empaqueta para reutilizarla

public class RegistroCuenta
{
   private int cuenta;
   private String primerNombre;
   private String apellidoPaterno;
   private double saldo;

   // el constructor sin argumentos llama a otro constructor con valores predeterminados
   public RegistroCuenta()
   {
      this( 0, "", "", 0.0 ); // llama al constructor con cuatro argumentos
   } // fin del constructor de RegistroCuenta sin argumentos

   // inicializa un registro
   public RegistroCuenta( int cta, String nombre, String apellido, double sal )
   {
      establecerCuenta( cta );
      establecerPrimerNombre( nombre );
      establecerApellidoPaterno( apellido );
      establecerSaldo( sal );
   } // fin del constructor de RegistroCuenta con cuatro argumentos

   // establece el número de cuenta
   public void establecerCuenta( int cta )
   {
      cuenta = cta;
   } // fin del método establecerCuenta

   // obtiene el número de cuenta
   public int obtenerCuenta()
   {
      return cuenta;
   } // fin del método obtenerCuenta

   // establece el primer nombre
   public void establecerPrimerNombre( String nombre )
   {
      primerNombre = nombre;
   } // fin del método establecerPrimerNombre

   // obtiene el primer nombre
   public String obtenerPrimerNombre()
   {
      return primerNombre;
   } // fin del método obtenerPrimerNombre

   // establece el apellido paterno
   public void establecerApellidoPaterno( String apellido )
   {
      apellidoPaterno = apellido;
   } // fin del método establecerApellidoPaterno

   // obtiene el apellido paterno
   public String obtenerApellidoPaterno()
   {
```

Figura 14.6 | `RegistroCuenta` mantiene la información para una cuenta. (Parte 1 de 2).

```
         return apellidoPaterno;
      } // fin del método obtenerApellidoPaterno

      // establece el saldo
      public void establecerSaldo( double sal )
      {
         saldo = sal;
      } // fin del método establecerSaldo

      // obtiene el saldo
      public double obtenerSaldo()
      {
         return saldo;
      } // fin del método obtenerSaldo
   } // fin de la clase RegistroCuenta
```

Figura 14.6 | `RegistroCuenta` mantiene la información para una cuenta. (Parte 2 de 2).

Observe que el directorio actual (que se especifica con **.**) se incluye en la ruta de clases. Esto asegura que el compilador pueda localizar otras clases en el mismo directorio que el de la clase que se está compilando. El separador de ruta que se utiliza en los comandos anteriores debe ser el apropiado para su plataforma; por ejemplo, un punto y coma (`;`) en Windows y dos puntos (`:`) en UNIX/Linux/MAC OS X.

Ahora examinaremos la clase `CrearArchivoTexto` (figura 14.7). La línea 14 declara la variable `Formatter` llamada `salida`. Como vimos en la sección 14.3, un objeto `Formatter` muestra en pantalla cadenas con formato, usando las mismas herramientas de formato que el método `System.out.printf`. Un objeto `Formatter` puede enviar datos a varias ubicaciones, como la pantalla o a un archivo, como lo hacemos aquí. El objeto `Formatter` se instancia en la línea 21, en el método `abrirArchivo` (líneas 17 a 34). El constructor que se utiliza en la línea 21 recibe un argumento: un objeto `String` que contiene el nombre del archivo, incluyendo su ruta. Si no se especifica una ruta, como se da aquí el caso, la JVM asume que los archivos están en el directorio desde el cual se ejecutó el programa. Para los archivos de texto, utilizamos la extensión `.txt`. Si el archivo no existe, se creará. Si se abre un archivo existente, su contenido se **trunca**; todos los datos en el archivo se descartan. En este punto, el archivo se abre para escritura y el objeto `Formatter` resultante se puede utilizar para escribir datos en el archivo. En las líneas 23 a 28 se maneja la excepción tipo `SecurityException`, que ocurre si el usuario no tiene permiso para escribir datos en el archivo. En las líneas 29 a 33 se maneja la excepción tipo `FileNotFoundException`, que ocurre si el archivo no existe y no se puede crear uno nuevo. Esta excepción también puede ocurrir si hay un error al abrir el archivo. Observe que en ambos manejadores de excepciones podemos llamar al método `static System.exit`, y pasarle el valor `1`. Este método termina la aplicación. Un argumento de `0` para el método `exit` indica la terminación exitosa del programa. Un valor distinto de cero, como el `1` en este ejemplo, por lo general, indica que ocurrió un error. Este valor se pasa a la ventana de comandos en la que se ejecutó el programa. El argumento es útil si el programa se ejecuta desde un **archivo de procesamiento por lotes** en los sistemas Windows, o una **secuencia de comandos de shell** en sistemas UNIX/Linux/Mac OS X. Los archivos de procesamiento por lotes y las secuencias de comandos de shell ofrecen una manera conveniente de ejecutar varios programas en secuencia. Cuando termina el primer programa, el siguiente programa empieza su ejecución. Es posible utilizar el argumento para el método `exit` en un archivo de procesamiento por lotes o secuencia de comandos de shell, para determinar si deben ejecutarse otros programas. Para obtener más información acerca de los archivos de procesamiento por lotes o las secuencias de comandos de shell, consulte la documentación de su sistema operativo.

El método `agregarRegistros` (líneas 37 a 91) pide al usuario que introduzca los diversos campos para cada registro, o la secuencia de teclas de fin de archivo cuando termine de introducir los datos. La figura 14.8 enlista las combinaciones de teclas para introducir el fin de archivo en varios sistemas computacionales.

En la línea 40 se crea un objeto `RegistroCuenta`, el cual se utilizará para almacenar los valores del registro actual introducido por el usuario. En la línea 42 se crea un objeto `Scanner` para leer la entrada del usuario mediante el teclado. En las líneas 44 a 48 y 50 a 52 se pide al usuario que introduzca los datos.

En la línea 54 se utiliza el método `hasNext` de `Scanner` para determinar si se ha introducido la combinación de teclas de fin de archivo. El ciclo se ejecuta hasta que `hasNext` encuentra los indicadores de fin de archivo.

```java
// Fig. 14.7: CrearArchivoTexto.java
// Uso de la clase Formatter para escribir datos en un archivo de texto.
import java.io.FileNotFoundException;
import java.lang.SecurityException;
import java.util.Formatter;
import java.util.FormatterClosedException;
import java.util.NoSuchElementException;
import java.util.Scanner;

import com.deitel.jhtp7.cap14.RegistroCuenta;

public class CrearArchivoTexto
{
   private Formatter salida; // objeto usado para enviar texto al archivo

   // permite al usuario abrir el archivo
   public void abrirArchivo()
   {
      try
      {
         salida = new Formatter( "clientes.txt" );
      } // fin de try
      catch ( SecurityException securityException )
      {
         System.err.println(
            "No tiene acceso de escritura a este archivo." );
         System.exit( 1 );
      } // fin de catch
      catch ( FileNotFoundException filesNotFoundException )
      {
         System.err.println( "Error al crear el archivo." );
         System.exit( 1 );
      } // fin de catch
   } // fin del método abrirArchivo

   // agrega registros al archivo
   public void agregarRegistros()
   {
      // objeto que se va a escribir en el archivo
      RegistroCuenta registro = new RegistroCuenta();

      Scanner entrada = new Scanner( System.in );

      System.out.printf( "%s\n%s\n%s\n%s\n\n",
         "Para terminar la entrada, escriba el indicador de fin de archivo ",
         "cuando se le pida que escriba los datos de entrada.",
         "En UNIX/Linux/Mac OS X escriba <ctrl> d y oprima Intro",
         "En Windows escriba <ctrl> z y oprima Intro" );

      System.out.printf( "%s\n%s",
         "Escriba el numero de cuenta (> 0), primer nombre, apellido paterno y saldo.",
         "? " );

      while ( entrada.hasNext() ) // itera hasta encontrar el indicador de fin de archivo
      {
         try // envía valores al archivo
         {
            // obtiene los datos que se van a enviar
            registro.establecerCuenta( entrada.nextInt() ); // lee el número de cuenta
```

Figura 14.7 | Creación de un archivo de texto secuencial. (Parte 1 de 2).

```
60                registro.establecerPrimerNombre( entrada.next() ); // lee el primer nombre
61                registro.establecerApellidoPaterno( entrada.next() ); // lee el apellido paterno
62                registro.establecerSaldo( entrada.nextDouble() ); // lee el saldo
63
64                if ( registro.obtenerCuenta() > 0 )
65                {
66                   // escribe el nuevo registro
67                   salida.format( "%d %s %s %.2f\n", registro.obtenerCuenta(),
68                      registro.obtenerPrimerNombre(), registro.obtenerApellidoPaterno(),
69                      registro.obtenerSaldo() );
70                } // fin de if
71                else
72                {
73                   System.out.println(
74                      "El numero de cuenta debe ser mayor que 0." );
75                } // fin de else
76             } // fin de try
77             catch ( FormatterClosedException formatterClosedException )
78             {
79                System.err.println( "Error al escribir en el archivo." );
80                return;
81             } // fin de catch
82             catch ( NoSuchElementException elementException )
83             {
84                System.err.println( "Entrada invalida. Intente de nuevo." );
85                entrada.nextLine(); // descarta la entrada para que el usuario intente de nuevo
86             } // fin de catch
87
88             System.out.printf( "%s %s\n%s", "Escriba el numero de cuenta (> 0),",
89                "primer nombre, apellido paterno y saldo.", "? " );
90          } // fin de while
91       } // fin del método agregarRegistros
92
93       // cierra el file
94       public void cerrarArchivo()
95       {
96          if ( salida != null )
97             salida.close();
98       } // fin del método cerrarArchivo
99    } // fin de la clase CrearArchivoTexto
```

Figura 14.7 | Creación de un archivo de texto secuencial. (Parte 2 de 2).

Sistema operativo	Combinación de teclas
UNIX/Linux/Mac OS X	*<Intro> <Ctrl> d*
Windows	*<Ctrl> z*

Figura 14.8 | Combinaciones de teclas de fin de archivo para diversos sistemas operativos.

En las líneas 59 a 62 se leen datos del usuario y se almacena la información del registro en el objeto `RegistroCuenta`. Cada instrucción lanza una excepción tipo `NoSuchElementException` (que se maneja en las líneas 82 a 86) si los datos se encuentran en el formato incorrecto (por ejemplo, una cadena cuando se espera un `int`), o si no hay más datos que introducir. Si el número de cuenta es mayor que 0 (línea 64), la información del registro

se escribe en `clientes.txt` (líneas 67 a 69) mediante el método `format`. Este método puede efectuar un formato idéntico al del método `System.out.printf`, que se utilizó en muchos de los ejemplos de capítulos anteriores. Este método envía una cadena con formato al destino de salida del objeto `Formatter`, en este caso el archivo `clientes.txt`. La cadena de formato `"%d &s &s &.2f\n"` indica que el registro actual se almacenará como un entero (el número de cuenta) seguido de una cadena (el primer nombre), otra cadena (el apellido paterno) y un valor de punto flotante (el saldo). Cada pieza de información se separa de la siguiente, mediante un espacio, y el valor tipo `double` (el saldo) se imprime en pantalla con dos dígitos a la derecha del punto decimal. Los datos en el archivo de texto se pueden ver con un editor, o posteriormente mediante un programa diseñado para leer el archivo (14.5.2) y obtener esos datos. Cuando se ejecutan las líneas 67 a 69, si se cierra el objeto `Formatter` se lanza una excepción tipo `FormatterClosedException` (que se maneja en las líneas 77 a 81). [*Nota:* también puede enviar datos a un archivo de texto mediante la clase `java.io.PrintWriter`, la cual también cuenta con el método `format` para enviar/imprimir datos con formato].

En las líneas 94 a 98 se declara el método `cerrarArchivo`, el cual cierra el objeto `Formatter` y el archivo de salida subyacente. En la línea 97 se cierra el objeto, mediante una llamada simple al método **`close`**. Si el método `close` no se llama en forma explícita, el sistema operativo comúnmente cierra el archivo cuando el programa termina de ejecutarse; éste es un ejemplo de las "tareas de mantenimiento" del sistema operativo.

La figura 14.9 ejecuta el programa. En la línea 8 se crea un objeto `CrearArchivoTexto`, el cual se utiliza posteriormente para abrir, agregar registros y cerrar el archivo (líneas 10 a 12). Los datos de ejemplo para esta aplicación se muestran en la figura 14.10. En la ejecución de ejemplo para este programa, el usuario introduce información para cinco cuentas, y después introduce el fin de archivo para indicar que ha terminado de introducir datos. La ejecución de ejemplo no muestra cómo aparecen realmente los registros de datos en el archivo. En la siguiente sección, para verificar que el archivo se haya creado sin problemas, presentamos un programa que lee el archivo e imprime su contenido. Como es un archivo de texto, también puede verificar la información abriendo el archivo en un editor de texto.

```java
1  // Fig. 14.9: PruebaCrearArchivoTexto.java
2  // Prueba de la clase CrearArchivoTexto.
3
4  public class PruebaCrearArchivoTexto
5  {
6     public static void main( String args[] )
7     {
8        CrearArchivoTexto aplicacion = new CrearArchivoTexto();
9
10       aplicacion.abrirArchivo();
11       aplicacion.agregarRegistros();
12       aplicacion.cerrarArchivo();
13    } // fin de main
14 } // fin de la clase PruebaCrearArchivoTexto
```

```
Para terminar la entrada, escriba el indicador de fin de archivo
cuando se le pida que escriba los datos de entrada.
En UNIX/Linux/Mac OS X escriba <ctrl> d y oprima Intro
En Windows escriba <ctrl> z y oprima Intro

Escriba el numero de cuenta (> 0), primer nombre, apellido paterno y saldo.
? 100 Bob Jones 24.98
Escriba el numero de cuenta (> 0), primer nombre, apellido paterno y saldo.
? 200 Steve Doe -345.67
Escriba el numero de cuenta (> 0), primer nombre, apellido paterno y saldo.
? 300 Pam White 0.00
Escriba el numero de cuenta (> 0), primer nombre, apellido paterno y saldo.
? 400 Sam Stone -42.16
```

Figura 14.9 | Prueba de la clase `CrearArchivoTexto`. (Parte 1 de 2).

```
Escriba el numero de cuenta (> 0), primer nombre, apellido paterno y saldo.
? 500 Sue Rich 224.62
Escriba el numero de cuenta (> 0), primer nombre, apellido paterno y saldo.
? ^Z
```

Figura 14.9 | Prueba de la clase `CrearArchivoTexto`. (Parte 2 de 2).

Datos de ejemplo			
100	Bob	Jones	24.98
200	Steve	Doe	-345.67
300	Pam	White	0.00
400	Sam	Stone	-42.16
500	Sue	Rich	224.62

Figura 14.10 | Datos de ejemplo para el programa de la figura 14.7.

14.5.2 Cómo leer datos de un archivo de texto de acceso secuencial

Los datos se almacenan en archivos, para poder procesarlos según sea necesario. En la sección 14.5.1 demostramos cómo crear un archivo para acceso secuencial. Esta sección muestra cómo leer los datos secuencialmente desde un archivo de texto. En esta sección, demostraremos cómo puede utilizarse la clase `Scanner` para recibir datos de un archivo, en vez del teclado.

La aplicación de las figuras 14.11 y 14.12 lee registros del archivo `"clientes.txt"` creado por la aplicación de la sección 14.5.1 y muestra el contenido de los registros. En la línea 13 de la figura 14.11 se declara un objeto `Scanner`, que se utilizará para obtener los datos de entrada del archivo.

El método `abrirArchivo` (líneas 16 a 27) abre el archivo en modo de lectura, creando una instancia de un objeto `Scanner` en la línea 20. Pasamos un objeto `File` al constructor, el cual especifica que el objeto `Scanner` leerá datos del archivo `"clientes.txt"` ubicado en el directorio desde el que se ejecuta la aplicación. Si no puede encontrarse el archivo, ocurre una excepción tipo `FileNotFoundException`. La excepción se maneja en las líneas 22 a 26.

```
1  // Fig. 14.11: LeerArchivoTexto.java
2  // Este programa lee un archivo de texto y muestra cada registro.
3  import java.io.File;
4  import java.io.FileNotFoundException;
5  import java.lang.IllegalStateException;
6  import java.util.NoSuchElementException;
7  import java.util.Scanner;
8
9  import com.deitel.jhtp7.cap14.RegistroCuenta;
10
11 public class LeerArchivoTexto
12 {
13    private Scanner entrada;
14
15    // permite al usuario abrir el archivo
16    public void abrirArchivo()
17    {
18       try
```

Figura 14.11 | Lectura de un archivo secuencial mediante un objeto `Scanner`. (Parte 1 de 2).

```
19          {
20             entrada = new Scanner( new File( "clientes.txt" ) );
21          } // fin de try
22          catch ( FileNotFoundException fileNotFoundException )
23          {
24             System.err.println( "Error al abrir el archivo." );
25             System.exit( 1 );
26          } // fin de catch
27       } // fin del método abrirArchivo
28
29       // lee registro del archivo
30       public void leerRegistros()
31       {
32          // objeto que se va a escribir en la pantalla
33          RegistroCuenta registro = new RegistroCuenta();
34
35          System.out.printf("%-9s%-15s%-18s%10s\n", “Cuenta”,
36             "Primer nombre", "Apellido paterno", "Saldo" );
37
38          try // lee registros del archivo, usando el objeto Scanner
39          {
40             while ( entrada.hasNext() )
41             {
42                registro.establecerCuenta( entrada.nextInt() ); // lee el número de cuenta
43                registro.establecerPrimerNombre( entrada.next() ); // lee el primer nombre
44                registro.establecerApellidoPaterno( entrada.next() ); // lee el apellido paterno
45                registro.establecerSaldo( entrada.nextDouble() ); // lee el saldo
46
47                // muestra el contenido del registro
48                System.out.printf( "<%-9d%-15s%-18s%10.2f\n",
49                   registro.obtenerCuenta(), registro.obtenerPrimerNombre(),
50                   registro.obtenerApellidoPaterno(), registro.obtenerSaldo() );
51             } // fin de while
52          } // fin de try
53          catch ( NoSuchElementException elementException )
54          {
55             System.err.println( "El archivo no esta bien formado." );
56             entrada.close();
57             System.exit( 1 );
58          } // fin de catch
59          catch ( IllegalStateException stateException )
60          {
61             System.err.println( "Error al leer del archivo." );
62             System.exit( 1 );
63          } // fin de catch
64       } // fin del método leerRegistros
65
66       // cierra el archivo y termina la aplicación
67       public void cerrarArchivo()
68       {
69          if ( entrada != null )
70             entrada.close(); // cierra el archivo
71       } // fin del método cerrarArchivo
72    } // fin de la clase LeerArchivoTexto
```

Figura 14.11 | Lectura de un archivo secuencial mediante un objeto `Scanner`. (Parte 2 de 2).

El método `leerRegistros` (líneas 30 a 64) lee y muestra registros del archivo. En la línea 33 se crea el objeto `RegistroCuenta` llamado `registro`, para almacenar la información del registro actual. En las líneas 35 y 36 se

```
1  // Fig. 14.12: PruebaLeerArchivoTexto.java
2  // Este programa prueba la clase LeerArchivoTexto.
3
4  public class PruebaLeerArchivoTexto
5  {
6     public static void main( String args[] )
7     {
8        LeerArchivoTexto aplicacion = new LeerArchivoTexto();
9
10       aplicacion.abrirArchivo();
11       aplicacion.leerRegistros();
12       aplicacion.cerrarArchivo();
13    } // fin de main
14 } // fin de la clase PruebaLeerArchivoTexto
```

```
Cuenta    Primer nombre   Apellido paterno   Saldo
100       Bob             Jones               24.98
200       Steve           Doe               -345.67
300       Pam             White                0.00
400       Sam             Stone              -42.16
500       Sue             Rich               224.62
```

Figura 14.12 | Prueba de la clase `LeerArchivoTexto`.

muestran encabezados para las columnas, en los resultados de la aplicación. En las líneas 40 a 51 se leen datos del archivo hasta llegar al marcador de fin de archivo (en cuyo caso, el método `hasNext` devolverá `false` en la línea 40). En las líneas 42 a 45 se utilizan los métodos `nextInt`, `next` y `nextDouble` de `Scanner` para recibir un entero (el número de cuenta), dos cadenas (el primer nombre y el apellido paterno) y un valor `double` (el saldo). Cada registro es una línea de datos en el archivo. Los valores se almacenan en el objeto `registro`. Si la información en el archivo no está bien formada (por ejemplo, que haya un apellido paterno en donde debe haber un saldo), se produce una excepción tipo `NoSuchElementException` al momento de introducir el registro. Esta excepción se maneja en las líneas 53 a 58. Si el objeto `Scanner` se cerró antes de introducir los datos, se produce una excepción tipo `IllegalStateException` (que se maneja en las líneas 59 a 63). Si no ocurren excepciones, la información del registro se muestra en pantalla (líneas 48 a 50). Observe en la cadena de formato de la línea 48 que el número de cuenta, primer nombre y apellido paterno están justificados a la izquierda, mientras que el saldo está justificado a la derecha y se imprime con dos dígitos de precisión. Cada iteración del ciclo introduce una línea de texto del archivo de texto, la cual representa un registro.

En las líneas 67 a 71 se define el método `cerrarArchivo`, el cual cierra el objeto `Scanner`. El método `main` se define en la figura 14.12, en las líneas 6 a 13. En la línea 8 se crea un objeto `LeerArchivoTexto`, el cual se utiliza entonces para abrir, agregar registros y cerrar el archivo (líneas 10 a 12).

14.5.3 Ejemplo práctico: un programa de solicitud de crédito

Para obtener datos secuencialmente de un archivo, por lo general, los programas empiezan leyendo desde el principio del archivo y leen todos los datos en forma consecutiva, hasta encontrar la información deseada. Podría ser necesario procesar el archivo secuencialmente varias veces (desde el principio del archivo) durante la ejecución de un programa. La clase `Scanner` no proporciona la habilidad de reposicionarse hasta el principio del archivo. Si es necesario leer el archivo de nuevo, el programa debe cerrar el archivo y volver a abrirlo.

El programa de las figuras 14.13 a 14.15 permite a un gerente de créditos obtener listas de clientes con saldos de cero (es decir, los clientes que no deben dinero), saldos con crédito (es decir, los clientes a quienes la compañía les debe dinero) y saldos con débito (es decir, los clientes que deben dinero a la compañía por los bienes y servicios recibidos en el pasado). Un saldo con crédito es un monto negativo, y un saldo con débito es un monto positivo.

Empezamos por crear un tipo `enum` (figura 14.13) para definir las distintas opciones del menú que tendrá el usuario. Las opciones y sus valores se enlistan en las líneas 7 a 10. El método `obtenerValor` (líneas 19 a 22) obtiene el valor de una constante `enum` específica.

```
// Fig. 14.13: OpcionMenu.java
// Define un tipo enum para las opciones del programa de consulta de crédito.

public enum OpcionMenu
{
   // declara el contenido del tipo enum
   SALDO_CERO( 1 ),
   SALDO_CREDITO( 2 ),
   SALDO_DEBITO( 3 ),
   FIN( 4 );

   private final int valor; // opción actual del menú

   OpcionMenu( int valorOpcion )
   {
      valor = valorOpcion;
   } // fin del constructor del tipo enum OpcionMenu

   public int obtenerValor()
   {
      return valor;
   } // fin del método obtenerValor
} // fin del tipo enum OpcionMenu
```

Figura 14.13 | Enumeración para las opciones del menú.

```
// Fig. 14.14: ConsultaCredito.java
// Este programa lee un archivo secuencialmente y muestra su
// contenido con base en el tipo de cuenta que solicita el usuario
// (saldo con crédito, saldo con débito o saldo de cero).
import java.io.File;
import java.io.FileNotFoundException;
import java.lang.IllegalStateException;
import java.util.NoSuchElementException;
import java.util.Scanner;

import com.deitel.jhtp7.cap14.RegistroCuenta;

public class ConsultaCredito
{
   private OpcionMenu tipoCuenta;
   private Scanner entrada;
   private OpcionMenu opciones[] = { OpcionMenu.SALDO_CERO,
      OpcionMenu.SALDO_CREDITO, OpcionMenu.SALDO_DEBITO,
      OpcionMenu.FIN };

   // lee los registros del archivo y muestra sólo los registros del tipo apropiado
   private void leerRegistros()
   {
      // objeto que se va a escribir en el archivo
      RegistroCuenta registro = new RegistroCuenta();

      try // lee registros
      {
         // abre el archivo para leer desde el principio
         entrada = new Scanner( new File( "clientes.txt") );

         while ( entrada.hasNext() ) // recibe los valores del archivo
```

Figura 14.14 | Programa de consulta de crédito. (Parte 1 de 3).

```
         {
            registro.establecerCuenta( entrada.nextInt() ); // lee número de cuenta
            registro.establecerPrimerNombre( entrada.next() ); // lee primer nombre
            registro.establecerApellidoPaterno( entrada.next() ); // lee apellido paterno
            registro.establecerSaldo( entrada.nextDouble() ); // lee saldo

            // si el tipo de cuenta es apropiado, muestra el registro
            if ( debeMostrar( registro.obtenerSaldo() ) )
               System.out.printf( "%-10d%-12s%-12s%10.2f\n",
                  registro.obtenerCuenta(), registro.obtenerPrimerNombre(),
                  registro.obtenerApellidoPaterno(), registro.obtenerSaldo() );
         } // fin de while
      } // fin de try
      catch ( NoSuchElementException elementException )
      {
         System.err.println( "El archivo no esta bien formado." );
         entrada.close();
         System.exit( 1 );
      } // fin de catch
      catch ( IllegalStateException stateException )
      {
         System.err.println( "Error al leer del archivo." );
         System.exit( 1 );
      } // fin de catch
      catch ( FileNotFoundException fileNotFoundException )
      {
         System.err.println( "No se puede encontrar el archivo." );
         System.exit( 1 );
      } // fin de catch
      finally
      {
         if ( entrada != null )
            entrada.close(); // cierra el objeto Scanner y el archivo
      } // fin de finally
   } // fin del método leerRegistros

   // usa el tipo de registro para determinar si el registro debe mostrarse
   private boolean debeMostrar( double saldo )
   {
      if ( ( tipoCuenta == OpcionMenu.SALDO_CREDITO )
         && ( saldo < 0 ) )
         return true;

      else if ( ( tipoCuenta == OpcionMenu.SALDO_DEBITO )
         && ( saldo > 0 ) )
         return true;

      else if ( ( tipoCuenta == OpcionMenu.SALDO_CERO )
         && ( saldo == 0 ) )
         return true;

      return false;
   } // fin del método debeMostrar

   // obtiene solicitud del usuario
   private OpcionMenu obtenerSolicitud()
   {
      Scanner textoEnt = new Scanner( System.in );
```

Figura 14.14 | Programa de consulta de crédito. (Parte 2 de 3).

```
      int solicitud = 1;

      // muestra opciones de solicitud
      System.out.printf( "\n%s\n%s\n%s\n%s\n%s\n",
         "Escriba solicitud", " 1 - Lista de cuentas con saldos de cero",
         " 2 - Lista de cuentas con saldos con credito",
         " 3 - Lista de cuentas con saldos con debito", " 4 - Finalizar ejecucion"  );

      try // trata de recibir la opción del menú
      {
         do // recibe solicitud del usuario
         {
            System.out.print( "\n? " );
            solicitud = textoEnt.nextInt();
         } while ( ( solicitud < 1 ) || ( solicitud > 4 ) );
      } // fin de try
      catch ( NoSuchElementException elementException )
      {
         System.err.println( "Entrada invalida." );
         System.exit( 1 );
      } // fin de catch

      return opciones[ solicitud - 1 ]; // devuelve valor de enum para la opción
   } // fin del método obtenerSolicitud

   public void procesarSolicitudes()
   {
      // obtiene la solicitud del usuario (saldo de cero, con crédito o con débito)
      tipoCuenta = obtenerSolicitud();

      while ( tipoCuenta != OpcionMenu.FIN )
      {
         switch ( tipoCuenta )
         {
            case SALDO_CERO:
               System.out.println( "nCuentas con saldos de cero:\n" );
               break;
            case SALDO_CREDITO:
               System.out.println( "\nCuentas con saldos con credito:\n" );
               break;
            case SALDO_DEBITO:
               System.out.println( "\nCuentas con saldos con debito:\n" );
               break;
         } // fin de switch

         leerRegistros();
         tipoCuenta = obtenerSolicitud();
      } // fin de while
   } // fin del método procesarSolicitudes
} // fin de la clase ConsultaCredito
```

Figura 14.14 | Programa de consulta de crédito. (Parte 3 de 3).

La figura 14.14 contiene la funcionalidad para el programa de consulta de crédito, y la figura 14.15 contiene el método `main` que ejecuta el programa. Este programa muestra un menú de texto y permite al gerente de créditos introducir una de tres opciones para obtener información sobre un crédito. La opción 1 (`SALDO_CERO`) produce una lista de cuentas con saldos de cero. La opción 2 (`SALDO_CREDITO`) produce una lista de cuentas con saldos con crédito. La opción 3 (`SALDO_DEBITO`) produce una lista de cuentas con saldos con débito. La opción 4 (`FIN`) termina la ejecución del programa. En la figura 14.16 se muestra un conjunto de resultados de ejemplo.

```java
// Fig. 14.15: PruebaConsultaCredito.java
// Este programa prueba la clase ConsultaCredito.

public class PruebaConsultaCredito
{
   public static void main( String args[] )
   {
      ConsultaCredito aplicacion = new ConsultaCredito();
      aplicacion.procesarSolicitudes();
   } // fin de main
} // fin de la clase PruebaConsultaCredito
```

Figura 14.15 | Prueba de la clase `ConsultaCredito`.

```
Escriba solicitud
 1 - Lista de cuentas con saldos de cero
 2 - Lista de cuentas con saldos con credito
 3 - Lista de cuentas con saldos con debito
 4 - Finalizar ejecucion

? 1

Cuentas con saldos de cero:

300       Pam         White             0.00

Escriba solicitud
 1 - Lista de cuentas con saldos de cero
 2 - Lista de cuentas con saldos con credito
 3 - Lista de cuentas con saldos con debito
 4 - Finalizar ejecucion

? 2

Cuentas con saldos con credito:

200       Steve       Doe            -345.67
400       Sam         Stone           -42.16

Escriba solicitud
 1 - Lista de cuentas con saldos de cero
 2 - Lista de cuentas con saldos con credito
 3 - Lista de cuentas con saldos con debito
 4 - Finalizar ejecucion

? 3

Cuentas con saldos con debito:

100       Bob         Jones            24.98
500       Sue         Rich            224.62

? 4
```

Figura 14.16 | Salida de ejemplo del programa de consulta de crédito de la figura 14.15.

Para recolectar la información de los registros, se lee todo el archivo completo y se determina si cada uno de los registro cumple o no con los criterios para el tipo de cuenta seleccionado por el gerente de créditos. El método `procesarSolicitudes` (líneas 116 a 139 de la figura 14.14) llama al método `obtenerSolicitud` para mostrar

las opciones del menú (línea 119) y almacena el resultado en la variable `OpcionMenu` llamada `tipoCuenta`. Observe que `obtenerSolicitud` traduce el número escrito por el usuario en un objeto `OpcionMenu`, usando el número para seleccionar un objeto `OpcionMenu` del arreglo `opciones`. En las líneas 121 a 138 se itera hasta que el usuario especifique que el programa debe terminar. La instrucción `switch` en las líneas 123 a 134 muestra un encabezado para imprimir el conjunto actual de registros en la pantalla. En la línea 136 se hace una llamada al método `leerRegistros` (líneas 22 a 67), el cual itera a través del archivo y lee todos los registros.

La línea 30 del método `leerRegistros` abre el archivo en modo de lectura con un objeto `Scanner`. Observe que el archivo se abrirá en modo de lectura con un nuevo objeto `Scanner` cada vez que se haga una llamada a este método, para que podamos leer de nuevo desde el principio del archivo. En las líneas 34 a 37 se lee un registro. En la línea 40 se hace una llamada al método `debeMostrar` (líneas 70 a 85), para determinar si el registro actual cumple con el tipo de cuenta solicitado. Si `debeMostrar` devuelve `true`, el programa muestra la información de la cuenta. Cuando se llega al marcador de fin de archivo, el ciclo termina y en la línea 65 se hace una llamada al método `close` de `Scanner` para cerrar el objeto `Scanner` y el archivo. Observe que esto ocurre en un bloque `finally`, el cual se ejecutará sin importar que se haya leído o no el archivo con éxito. Una vez que se hayan leído todos los registros, el control regresa al método `procesarSolicitudes` y se hace una llamada otra vez al método `obtenerSolicitud` (línea 137) para obtener la siguiente opción de menú del usuario. La figura 14.15 contiene el método `main`, y llama al método `procesarSolicitudes` en la línea 9.

14.5.4 Actualización de archivos de acceso secuencial

En muchos archivos secuenciales, los datos no se pueden modificar sin el riesgo de destruir otros datos en el archivo. Por ejemplo, si el nombre “`White`” tuviera que cambiarse a “`Worthington`”, el nombre anterior no podría simplemente sobrescribirse, debido a que el nuevo nombre requiere más espacio. El registro para `White` se escribió en el archivo como

```
300 Pam White 0.00
```

Si el registro se sobrescribe empezando en la misma ubicación en el archivo que utiliza el nuevo nombre, el registro será

```
300 Pam Worthington 0.00
```

El nuevo registro es más extenso (tiene más caracteres) que el registro original. Los caracteres más allá de la segunda “`o`” en “`Worthington`” sobrescribirán el principio del siguiente registro secuencial en el archivo. El problema aquí es que los campos en un archivo de texto (y por ende, los registros) pueden variar en tamaño. Por ejemplo, 7, 14, -117, 2074 y 27383 son todos valores `int` almacenados en el mismo número de bytes (4) internamente, pero son campos con distintos tamaños cuando se muestran en la pantalla, o se escriben en un archivo como texto.

Por lo tanto, los registros en un archivo de acceso secuencial comúnmente no se actualizan por partes. En vez de ello, generalmente se sobrescribe todo el archivo. Para realizar el cambio anterior, los registros antes de `300 Pam White 0.00` se copian a un nuevo archivo, se escribe el nuevo registro (que puede tener un tamaño distinto al que está sustituyendo) y se copian los registros después de `300 Pam White 0.00` al nuevo archivo. Es inconveniente actualizar sólo un registro, pero razonable si una porción substancial de los registros necesitan actualización.

14.6 Serialización de objetos

En la sección 14.5 demostramos cómo escribir los campos individuales de un objeto `RegistroCuenta` en un archivo como texto, y cómo leer esos campos de un archivo y colocar sus valores en un objeto `RegistroCuenta` en la memoria. En los ejemplos, se usó `RegistroCuenta` para agregar la información de un registro. Cuando las variables de instancia de un objeto `RegistroCuenta` se enviaban a un archivo en disco, se perdía cierta información, como el tipo de cada valor. Por ejemplo, si se lee el valor `"3"` de un archivo, no hay forma de saber si el valor proviene de un `int`, un `String` o un `double`. En un disco sólo tenemos los datos, no la información sobre los tipos. Si el programa que va a leer estos datos “sabe” a qué tipo de objeto corresponden, entonces simplemente se leen y se colocan en objetos de ese tipo. Por ejemplo, en la sección 14.5.2 sabemos que introduciremos un `int` (el número de cuenta), seguido de dos objetos `String` (el primer nombre y el apellido paterno) y un `double` (el saldo). También sabemos que estos valores se separan mediante espacios, y sólo se coloca un registro en cada línea. Algunas veces no sabremos con exactitud cómo se almacenan los datos en un archivo. En tales casos, sería

conveniente poder escribir o leer un objeto completo de un archivo. Java cuenta con dicho mecanismo, llamado **serialización de objetos**. Un **objeto serializado** es un objeto que se representa como una secuencia de bytes, la cual incluye los datos del objeto, así como información acerca del tipo del objeto y los tipos de los datos almacenados en el mismo. Una vez que se escribe un objeto serializado en un archivo, se puede leer de ese archivo y **deserializarse**; es decir, la información del tipo y los bytes que representan al objeto y sus datos se puede utilizar para recrear el objeto en memoria.

Las clases `ObjectInputStream` y `ObjectOutputStream`, que implementan en forma respectiva a las interfaces `ObjectInput` y `ObjectOutput`, permiten leer/escribir objetos completos de/en un flujo (posiblemente un archivo). Para utilizar la serialización con los archivos, inicializamos los objetos `ObjectInputStream` y `ObjectOutputStream` con objetos flujo que pueden leer y escribir información desde/hacia los archivos; objetos de las clases `FileInputStream` y `FileOutputStream`, respectivamente. La acción de inicializar objetos flujo con otros objetos flujo de esta forma se conoce algunas veces como **envoltura**: el nuevo objeto flujo que se va a crear envuelve al objeto flujo especificado como un argumento del constructor. Por ejemplo, para envolver un objeto `FileInputStream` en un objeto `ObjectInputStream`, pasamos el objeto `FileInputStream` al constructor de `ObjectInputStream`.

La interfaz `ObjectOutput` contiene el método **`writeObject`**, el cual toma un objeto `Object` que implementa a la interfaz `Serializable` (que veremos en breve) como un argumento y escribe su información a un objeto `OutputStream`. De manera correspondiente, la interfaz `ObjectInput` contiene el método **`readObject`**, el cual lee y devuelve una referencia a un objeto `Object` de un objeto `InputStream`. Una vez que se lee un objeto, su referencia puede convertirse en el tipo actual del objeto. Como veremos en el capítulo 24, Redes, las aplicaciones que se comunican a través de una red (como Internet) también pueden transmitir objetos completos a través de la red.

En esta sección vamos a crear y manipular archivos de acceso secuencial, usando la serialización de objetos. Ésa se realiza mediante flujos basados en bytes, de manera que los archivos secuenciales que se creen y manipulen serán archivos binarios. Recuerde que los archivos binarios no se pueden ver en los editores de texto estándar. Por esta razón, escribimos una aplicación separada que sabe cómo leer y mostrar objetos serializados.

14.6.1 Creación de un archivo de acceso secuencial mediante el uso de la serialización de objetos

Empezaremos por crear y escribir objetos serializados a un archivo de acceso secuencial. En esta sección reutilizamos la mayor parte del código de la sección 14.5, por lo que sólo nos enfocaremos en las nuevas características.

Definición de la clase `RegistroCuentaSerializable`

Para empezar, modificaremos nuestra clase `RegistroCuenta`, de manera que los objetos de esta clase puedan serializarse. La clase `RegistroCuentaSerializable` (figura 14.17) implementa a la interfaz `Serializable` (línea 7), la cual permite serializar y deserializar los objetos de la clase `RegistroCuentaSerializable` con objetos `ObjectOutputStream` y `ObjectInputStream`. La interfaz `Serializable` es una **interfaz de marcado**. Dicha interfaz no contiene métodos. Una clase que implementa a **`Serializable`** se marca como objeto `Serializable`. Esto es importante, ya que un objeto `ObjectOutputStream` no enviará un objeto como salida a menos que *sea un* objeto `Serializable`, lo cual es el caso para cualquier objeto de una clase que implemente a `Serializable`.

```
1  // Fig. 14.17: RegistroCuentaSerializable.java
2  // Una clase que representa un registro de información.
3  package com.deitel.jhtp7.cap14; // empaquetada para reutilizarla
4
5  import java.io.Serializable;
6
7  public class RegistroCuentaSerializable implements Serializable
8  {
9     private int cuenta;
10    private String primerNombre;
```

Figura 14.17 | La clase `RegistroCuentaSerializable` para los objetos serializables. (Parte 1 de 3).

```
   private String apellidoPaterno;
   private double saldo;

   // el constructor sin argumentos llama al otro constructor con valores predeterminados
   public RegistroCuentaSerializable()
   {
      this( 0, "", "", 0.0 );
   } // fin del constructor de RegistroCuentaSerializable sin argumentos

   // el constructor con cuatro argumentos inicializa un registro
   public RegistroCuentaSerializable(
      int cta, String nombre, String apellido, double sal )
   {
      establecerCuenta( cta );
      establecerPrimerNombre( nombre );
      establecerApellidoPaterno( apellido );
      establecerSaldo( sal );
   } // fin del constructor de RegistroCuentaSerializable con cuatro argumentos

   // establece el número de cuenta
   public void establecerCuenta( int cta )
   {
      cuenta = cta;
   } // fin del método establecerCuenta

   // obtiene el número de cuenta
   public int obtenerCuenta()
   {
      return cuenta;
   } // fin del método obtenerCuenta

   // establece el primer nombre
   public void establecerPrimerNombre( String nombre )
   {
      primerNombre = nombre;
   } // fin del método establecerPrimerNombre

   // obtiene el primer nombre
   public String obtenerPrimerNombre()
   {
      return primerNombre;
   } // fin del método obtenerPrimerNombre

   // establece el apellido paterno
   public void establecerApellidoPaterno( String apellido )
   {
      apellidoPaterno = apellido;
   } // fin del método establecerApellidoPaterno

   // obtiene el apellido paterno
   public String obtenerApellidoPaterno()
   {
      return apellidoPaterno;
   } // fin del método obtenerApellidoPaterno

   // establece el saldo
   public void establecerSaldo( double sal )
   {
      saldo = sal;
```

Figura 14.17 | La clase `RegistroCuentaSerializable` para los objetos serializables. (Parte 2 de 3).

```
   } // fin del método establecerSaldo

   // obtiene el saldo
   public double obtenerSaldo()
   {
      return saldo;
   } // fin del método obtenerSaldo
} // fin de la clase RegistroCuentaSerializable
```

Figura 14.17 | La clase `RegistroCuentaSerializable` para los objetos serializables. (Parte 3 de 3).

En una clase que implementa a `Serializable`, el programador debe asegurar que cada variable de instancia de la clase sea de un tipo `Serializable`. Cualquier variable de instancia que no sea serializable debe declararse como **`transient`**, para indicar que no es `Serializable` y debe ignorarse durante el proceso de serialización. De manera predeterminada, todas las variables de tipos primitivos son serializables. Para las variables de tipos de referencias, debe comprobar la definición de la clase (y posiblemente de sus superclases) para asegurar que el tipo sea `Serializable`. De manera predeterminada, los objetos tipo arreglo son serializables. No obstante, si el arreglo contiene referencias a otros objetos, éstos pueden o no ser serializables.

La clase `RegistroCuentaSerializable` contiene los miembros de datos `private` llamados `cuenta`, `primerNombre`, `apellidoPaterno` y `saldo`. Esta clase también proporciona métodos `public` *establecer* y *obtener* para acceder a los campos `private`.

Ahora hablaremos sobre el código que crea el archivo de acceso secuencial (figuras 14.18 y 14.19). Aquí nos concentraremos sólo en los nuevos conceptos. Como dijimos en la sección 14.3, un programa puede abrir un archivo creando un objeto de las clases de flujo `FileInputStream` o `FileOuptutStream`. En este ejemplo, el archivo se abrirá en modo de salida, por lo que el programa crea un objeto `FileOutputStream` (línea 21 de la figura 14.18). El argumento de cadena que se pasa al constructor de `FileOutputStream` representa el nombre y la ruta del archivo que se va a abrir. Los archivos existentes que se abren en modo de salida de esta forma se truncan. Observe que se utiliza la extensión de archivo `.ser`; utilizamos esta extensión de archivo para los archivos binarios que contienen objetos serializados.

```
1  // Fig. 14.18: CrearArchivoSecuencial.java
2  // Escritura de objetos en forma secuencial a un archivo, con la clase ObjectOutputStream.
3  import java.io.FileOutputStream;
4  import java.io.IOException;
5  import java.io.ObjectOutputStream;
6  import java.util.NoSuchElementException;
7  import java.util.Scanner;
8  
9  import com.deitel.jhtp7.cap14.RegistroCuentaSerializable;
10 
11 public class CrearArchivoSecuencial
12 {
13    private ObjectOutputStream salida; // envía los datos a un archivo
14 
15    // permite al usuario especificar el nombre del archivo
16    public void abrirArchivo()
17    {
18       try // abre el archivo
19       {
20          salida = new ObjectOutputStream(
21             new FileOutputStream( "clientes.ser" ) );
22       } // fin de try
23       catch ( IOException ioException )
```

Figura 14.18 | Archivo secuencial creado mediante `ObjectOutputStream`. (Parte 1 de 3).

```
      {
         System.err.println( "Error al abrir el archivo." );
      } // fin de catch
   } // fin del método abrirArchivo

   // agrega registros al archivo
   public void agregarRegistros()
   {
      RegistroCuentaSerializable registro; // objeto que se va a escribir al archivo
      int numeroCuenta = 0; // número de cuenta para el objeto registro
      String primerNombre; // primer nombre para el objeto registro
      String apellidoPaterno; // apellido paterno para el objeto registro
      double saldo; // saldo para el objeto registro

      Scanner entrada = new Scanner( System.in );

      System.out.printf( "%s\n%s\n%s\n%s\n\n",
         "Para terminar de introducir datos, escriba el indicador de fin de archivo",
         "Cuando se le pida que introduzca los datos.",
         "En UNIX/Linux/Mac OS X escriba <ctrl> d y oprima Intro",
         "En Windows escriba <ctrl> z y oprima Intro" );

      System.out.printf( "%s\n%s",
         "Escriba el numero de cuenta (> 0), primer nombre, apellido y saldo.",
         "? " );

      while ( entrada.hasNext() ) // itera hasta el indicador de fin de archivo
      {
         try // envía los valores al archivo
         {
            numeroCuenta = entrada.nextInt(); // lee el número de cuenta
            primerNombre = entrada.next(); // lee el primer nombre
            apellidoPaterno = entrada.next(); // lee el apellido paterno
            saldo = entrada.nextDouble(); // lee el saldo

            if ( numeroCuenta > 0 )
            {
               // crea un registro nuevo
               registro = new RegistroCuentaSerializable( numeroCuenta,
                  primerNombre, apellidoPaterno, saldo );
               salida.writeObject( registro ); // envía el registro como salida
            } // fin de if
            else
            {
               System.out.println(
                  "El numero de cuenta debe ser mayor de 0." );
            } // fin de else
         } // fin de try
         catch ( IOException ioException )
         {
            System.err.println( "Error al escribir en el archivo." );
            return;
         } // fin de catch
         catch ( NoSuchElementException elementException )
         {
            System.err.println( "Entrada invalida. Intente de nuevo." );
            entrada.nextLine(); // descarta la entrada para que el usuario intente de
            nuevo
         } // fin de catch
```

Figura 14.18 | Archivo secuencial creado mediante `ObjectOutputStream`. (Parte 2 de 3).

```

          System.out.printf( "%s %s\n%s", "Escriba el numero de cuenta (>0),",
             "primer nombre, apellido y saldo.", "? " );
       } // fin de while
    } // fin del método agregarRegistros

    // cierra el archivo y termina la aplicación
    public void cerrarArchivo()
    {
       try // cierra el archivo
       {
          if ( salida != null )
             salida.close();
       } // fin de try
       catch ( IOException ioException )
       {
          System.err.println( "Error al cerrar el archivo." );
          System.exit( 1 );
       } // fin de catch
    } // fin del método cerrarArchivo
} // fin de la clase CrearArchivoSecuencial
```

Figura 14.18 | Archivo secuencial creado mediante `ObjectOutputStream`. (Parte 3 de 3).

```
// Fig. 14.19: PruebaCrearArchivoSecuencial.java
// Prueba de la clase CrearArchivoSecuencial.

public class PruebaCrearArchivoSecuencial
{
   public static void main( String args[] )
   {
      CrearArchivoSecuencial aplicacion = new CrearArchivoSecuencial();

      aplicacion.abrirArchivo();
      aplicacion.agregarRegistros();
      aplicacion.cerrarArchivo();
   } // fin de main
} // fin de la clase PruebaCrearArchivoSecuencial
```

```
Para terminar de introducir datos, escriba el indicador de fin de archivo
cuando se le pida que introduzca los datos.
En UNIX/Linux/Mac OS X escriba <ctrl> d y oprima Intro
En Windows escriba <ctrl> z y oprima Intro

Escriba el numero de cuenta (> 0), primer nombre, apellido y saldo.
? 100 Bob Jones 24.98
Escriba el numero de cuenta (> 0), primer nombre, apellido y saldo.
? 200 Steve Doe -345.67
Escriba el numero de cuenta (> 0), primer nombre, apellido y saldo.
? 300 Pam White 0.00
Escriba el numero de cuenta (> 0), primer nombre, apellido y saldo.
? 400 Sam Stone -42.16
Escriba el numero de cuenta (> 0), primer nombre, apellido y saldo.
? 500 Sue Rich 224.62
Escriba el numero de cuenta (> 0), primer nombre, apellido y saldo.
? ^Z
```

Figura 14.19 | Prueba de la clase `CrearArchivoSecuencial`.



Error común de programación 14.2

Es un error lógico abrir un archivo existente en modo de salida cuando, de hecho, el usuario desea preservar ese archivo.

La clase `FileOutputStream` cuenta con métodos para escribir arreglos tipo `byte` y objetos `byte` individuales en un archivo. En este programa deseamos escribir objetos en un archivo; una capacidad que no proporciona `FileOutputStream`. Por esta razón, envolvemos un objeto `FileOutputStream` en un objeto `ObjectOutputStream`, pasando el nuevo objeto `FileOutputStream` al constructor de `ObjectOutputStream` (líneas 20 y 21). El objeto `ObjectOutputStream` utiliza al objeto `FileOutputStream` para escribir objetos en el archivo. En las líneas 20 y 21 se podría lanzar una excepción tipo `IOException` si ocurre un problema al abrir el archivo (por ejemplo, cuando se abre un archivo para escribir en una unidad de disco con espacio insuficiente, o cuando se abre un archivo de sólo lectura para escribir datos). Si es así, el programa muestra un mensaje de error (líneas 23 a 26). Si no ocurre una excepción, el archivo se abre y se puede utilizar la variable `salida` para escribir objetos en el archivo.

Este programa asume que los datos se introducen de manera correcta y en el orden de número de registro apropiado. El método `agregarRegistros` (líneas 30 a 86) realiza la operación de escritura. En las líneas 62 y 63 se crea un objeto `RegistroCuentaSerializable` a partir de los datos introducidos por el usuario. En la línea 64 se hace una llamada al método `writeObject` de `ObjectOutputStream` para escribir el objeto `registro` en el archivo de salida. Observe que sólo se requiere una instrucción para escribir todo el objeto.

El método `cerrarArchivo` (líneas 89 a 101) cierra el archivo. Este método llama al método `close` de `ObjectOutputStream` en `salida` para cerrar el objeto `ObjectOutputStream` y su objeto `FileOutputStream` subyacente (línea 94). Observe que la llamada al método `close` está dentro de un bloque `try`. El método `close` lanza una excepción `IOException` si el archivo no se puede cerrar en forma apropiada. En este caso, es importante notificar al usuario que la información en el archivo podría estar corrupta. Al utilizar flujos envueltos, si se cierra el flujo exterior también se cierra el archivo subyacente.

En la ejecución de ejemplo para el programa de la figura 14.19, introdujimos información para cinco cuentas; la misma información que se muestra en la figura 14.10. El programa no muestra cómo aparecen realmente los registros en el archivo. Recuerde que ahora estamos usando archivos binarios, que no pueden ser leídos por los humanos. Para verificar que el archivo se haya creado exitosamente, la siguiente sección presenta un programa para leer el contenido del archivo.

14.6.2 Lectura y deserialización de datos de un archivo de acceso secuencial

Como vimos en la sección 14.5.2, los datos se almacenan en archivos, para que puedan obtenerse y procesarse según sea necesario. En la sección anterior mostramos cómo crear un archivo para acceso secuencial, usando la serialización de objetos. En esta sección, veremos cómo leer datos serializados de un archivo, en forma secuencial.

El programa de las figuras 14.20 y 14.21 lee registros de un archivo creado por el programa de la sección 14.6.1 y muestra el contenido. El programa abre el archivo en modo de entrada, creando un objeto `FileInputStream` (línea 21). El nombre del archivo a abrir se especifica como un argumento para el constructor de `FileInputStream`. En la figura 14.18 escribimos objetos al archivo, usando un objeto `ObjectOutputStream`. Los datos se deben leer del archivo en el mismo formato en el que se escribió. Por lo tanto, utilizamos un objeto `ObjectInputStream` envuelto alrededor de un objeto `FileInputStream` en este programa (líneas 20 y 21). Si no ocurren excepciones al abrir el archivo, podemos usar la variable `entrada` para leer objetos del archivo.

El programa lee registros del archivo en el método `leerRegistros` (líneas 30 a 60). En la línea 40 se hace una llamada al método `readObject` de `ObjectInputStream` para leer un objeto `Object` del archivo. Para utilizar los métodos específicos de `RegistroCuentaSerializable`, realizamos una conversión descendente en el objeto `Object` devuelto, al tipo `RegistroCuentaSerializable`. El método `readObject` lanza una excepción tipo `EOFException` (que se procesa en las líneas 48 a 51) si se hace un intento por leer más allá del fin del archivo. El método `readObject` lanza una excepción `ClassNotFoundException` si no se puede localizar la clase para el objeto que se está leyendo. Esto podría ocurrir si se accede al archivo en una computadora que no tenga esa clase. La figura 14.21 contiene el método `main` (líneas 6 a 13), el cual abre el archivo, llama al método `leerRegistros` y cierra el archivo.

```java
// Fig. 14.20: LeerArchivoSecuencial.java
// Este programa lee un archivo de objetos en forma secuencial
// y muestra cada registro.
import java.io.EOFException;
import java.io.FileInputStream;
import java.io.IOException;
import java.io.ObjectInputStream;

import com.deitel.jhtp7.cap14.RegistroCuentaSerializable;

public class LeerArchivoSecuencial
{
   private ObjectInputStream entrada;

   // permite al usuario seleccionar el archivo a abrir
   public void abrirArchivo()
   {
      try // abre el archivo
      {
         entrada = new ObjectInputStream(
            new FileInputStream( "clientes.ser" ) );
      } // fin de try
      catch ( IOException ioException )
      {
         System.err.println( "Error al abrir el archivo." );
      } // fin de catch
   } // fin del método abrirArchivo

   // lee el registro del archivo
   public void leerRegistros()
   {
      RegistroCuentaSerializable registro;
      System.out.printf( "%-10s%-15s%-15s%10s\n", "Cuenta",
         "Primer nombre", "Apellido paterno", "Saldo" );

      try // recibe los valores del archivo
      {
         while ( true )
         {
            registro = ( RegistroCuentaSerializable ) entrada.readObject();

            // muestra el contenido del registro
            System.out.printf( "%-10d%-15s%-15s%11.2f\n",
               registro.obtenerCuenta(), registro.obtenerPrimerNombre(),
               registro.obtenerApellidoPaterno(), registro.obtenerSaldo() );
         } // fin de while
      } // fin de try
      catch ( EOFException endOfFileException )
      {
         return; // se llegó al fin del archivo
      } // fin de catch
      catch ( ClassNotFoundException classNotFoundException )
      {
         System.err.println( "No se pudo crear el objeto." );
      } // fin de catch
      catch ( IOException ioException )
      {
         System.err.println( "Error al leer el archivo." );
      } // fin de catch
```

Figura 14.20 | Lectura de un archivo secuencial, usando un objeto `ObjectInputStream`. (Parte 1 de 2).

```
   } // fin del método leerRegistros

   // cierra el archivo y termina la aplicación
   public void cerrarArchivo()
   {
      try // cierra el archivo y sale
      {
         if ( entrada != null )
            entrada.close();
         System.exit( 0 );
      } // fin de try
      catch ( IOException ioException )
      {
         System.err.println( "Error al cerrar el archivo." );
         System.exit( 1 );
      } // fin de catch
   } // fin del método cerrarArchivo
} // fin de la clase LeerArchivoSecuencial
```

Figura 14.20 | Lectura de un archivo secuencial, usando un objeto `ObjectInputStream`. (Parte 2 de 2).

```
// Fig. 14.21: PruebaLeerArchivoSecuencial.java
// Este programa prueba la clase ReadSequentialFile.

public class PruebaLeerArchivoSecuencial
{
   public static void main( String args[] )
   {
      LeerArchivoSecuencial aplicacion = new LeerArchivoSecuencial();

      aplicacion.abrirArchivo();
      aplicacion.leerRegistros();
      aplicacion.cerrarArchivo();
   } // fin de main
} // fin de la clase PruebaLeerArchivoSecuencial
```

```
Cuenta   Primer nombre   Apellido paterno    Saldo
100      Bob             Jones                 24.98
200      Steve           Doe                 -345.67
300      Pam             White                  0.00
400      Sam             Stone                -42.16
500      Sue             Rich                 224.62
```

Figura 14.21 | Prueba de la clase `LeerArchivoSecuencial`.

14.7 Clases adicionales de `java.io`

Ahora le presentaremos otras clases útiles en el paquete `java.io`. Veremos las generalidades acerca de las interfaces y clases adicionales para los flujos de entrada y salida basados en bytes, y los flujos de entrada y salida basados en caracteres.

Interfaces y clases para la entrada y salida basadas en bytes

`InputStream` y `OutputStream` (subclases de `Object`) son clases `abstract` que declaran métodos para realizar operaciones basadas en bytes de entrada y salida, respectivamente. En este capítulo utilizamos las clases concretas `FileInputStream` (una subclase de `InputStream`) y `FileOutputStream` (una subclase de `OutputStream`) para manipular archivos.

Las **canalizaciones** son canales de comunicación sincronizados entre subprocesos; hablaremos sobre los subprocesos en el capítulo 23, Subprocesamiento múltiple. Java proporciona las clases PipedOutputStream (una subclase de OutputStream) y PipedInputStream (una subclase de InputStream) para establecer canalizaciones entre dos subprocesos en un programa. Un subproceso envía datos a otro, escribiendo a un objeto PipedOutputStream. El subproceso de destino lee la información de la canalización mediante un objeto PipedInputStream.

Un objeto FilterInputStream **filtra** a un objeto InputStream, y un objeto FilterOutputStream filtra a un objeto OutputStream. Filtrar significa simplemente que el flujo que actúa como filtro proporciona una funcionalidad adicional, como la agregación de bytes de datos en unidades de tipo primitivo significativas. FilterInputStream y FilterOutputStream son clases abstract, por lo que sus subclases concretas proporcionan sus capacidades de filtrado.

Un objeto PrintStream (una subclase de FilterOutputStream) envía texto como salida hacia el flujo especificado. En realidad, hemos estado utilizando la salida mediante PrintStream a lo largo de este texto, hasta este punto; System.out y System.err son objetos PrintStream.

Leer datos en forma de bytes sin ningún formato es un proceso rápido, pero crudo. Por lo general, los programas leen datos como agregados de bytes que forman un valor int, un float, un double y así, sucesivamente. Los programas de Java pueden utilizar varias clases para recibir datos de entrada y enviar datos de salida en forma de agregación.

La interfaz DataInput describe métodos para leer tipos primitivos desde un flujo de entrada. Las clases DataInputStream y RandomAccessFile implementan a esta interfaz para leer conjuntos de bytes y verlos como valores de tipo primitivo. La interfaz DataInput incluye los métodos readLine (para arreglos byte), readBoolean, readByte, readChar, readDouble, readFloat, readFully (para arreglos byte), readInt, readLong, readShort, readUnsignedByte, readUnsignedShort, readUTF (para leer caracteres Unicode codificados por Java; hablaremos sobre la codificación UTF en el apéndice I, Unicode®) y skipBytes.

La interfaz DataOutput describe un conjunto de métodos para escribir tipos primitivos hacia un flujo de salida. Las clases DataOutputStream (una subclase de FilterOutputStream) y RandomAccessFile implementan a esta interfaz para escribir valores de tipos primitivos como bytes. La interfaz DataOutput incluye versiones sobrecargadas del método write (para un byte o para arreglo byte) y los métodos writeBoolean, writeByte, writeBytes, writeChar, writeChars (para objetos String Unicode), writeDouble, writeFloat, writeInt, writeLong, writeShort y writeUTF (para enviar texto modificado para Unicode).

El **uso de un búfer** es una técnica para mejorar el rendimiento de las operaciones de E/S. Con un objeto BufferedOutputStream (una subclase de la clase FilterOutputStream), cada instrucción de salida no produce necesariamente una transferencia física real de datos hacia el dispositivo de salida (una operación lenta, en comparación con las velocidades del procesador y de la memoria principal). En vez de ello, cada operación de salida se dirige hacia una región en memoria conocida como **búfer**, que es lo suficientemente grande como para almacenar los datos de muchas operaciones de salida. Después, la transferencia real hacia el dispositivo de salida se realiza en una sola **operación física de salida** extensa cada vez que se llena el búfer. Las operaciones de salida dirigidas hacia el búfer de salida en memoria se conocen a menudo como **operaciones lógicas de salida**. Con un objeto BufferedOutputStream, se puede forzar a un búfer parcialmente lleno para que envíe su contenido al dispositivo en cualquier momento, mediante la invocación del método flush del objeto flujo.

El uso de búfer puede aumentar considerablemente la eficiencia de una aplicación. Las operaciones comunes de E/S son extremadamente lentas, en comparación con la velocidad de acceso de la memoria de la computadora. El uso de búfer reduce el número de operaciones de E/S, al combinar primero las operaciones de salida más pequeñas en la memoria. El número de operaciones físicas de E/S reales es pequeño, en comparación con el número de solicitudes de E/S emitidas por el programa. Por ende, el programa que usa un búfer es más eficiente.



Tip de rendimiento 14.1

La E/S con búfer puede producir mejoras considerables en el rendimiento, en comparación con la E/S sin búfer.

Con un objeto BufferedInputStream (una subclase de la clase FilterInputStream), muchos trozos "lógicos" de datos de un archivo se leen como una sola **operación física de entrada** extensa y se envían a un búfer de memoria. A medida que un programa solicita cada nuevo trozo de datos, éste se toma del búfer. (A este procedimiento se le conoce como **operación lógica de entrada**). Cuando el búfer está vacío, se lleva a cabo la siguiente operación física de entrada real desde el dispositivo de entrada, para leer el siguiente grupo de trozos "lógicos" de

datos. Por lo tanto, el número de operaciones físicas de entrada reales es pequeño, en comparación con el número de solicitudes de lectura emitidas por el programa.

La E/S de flujos en Java incluye herramientas para recibir datos de entrada de arreglos `byte` en memoria, y enviar datos de salida a arreglos `byte` en memoria. Un objeto `ByteArrayInputStream` (una subclase de `InputStream`) lee de un arreglo `byte` en memoria. Un objeto `ByteArrayOutputStream` (una subclase de `OutputStream`) escribe en un arreglo `byte` en memoria. Una aplicación de la E/S con arreglos `byte` es la validación de datos. Un programa puede recibir como entrada una línea completa a la vez desde el flujo de entrada, para colocarla en un arreglo `byte`. Después puede usarse una rutina de validación para analizar detalladamente el contenido del arreglo `byte` y corregir los datos, si es necesario. Finalmente, el programa puede recibir los datos de entrada del arreglo `byte`, "sabiendo" que los datos de entrada se encuentran en el formato adecuado. Enviar datos de salida a un arreglo `byte` es una excelente manera de aprovechar las poderosas herramientas de formato para los datos de salida que proporcionan los flujos en Java. Por ejemplo, los datos pueden almacenarse en un arreglo `byte`, utilizando el mismo formato que se mostrará posteriormente, y luego el arreglo `byte` se puede enviar hacia un archivo en disco para preservar la imagen en pantalla.

Un objeto `SequenceInputStream` (una subclase de `InputStream`) permite la concatenación de varios objetos `InputStream`, por lo que el programa ve al grupo como un flujo `InputStream` continuo. Cuando el programa llega al final de un flujo de entrada, ese flujo se cierra y se abre el siguiente flujo en la secuencia.

Interfaces y clases para la entrada y salida basadas en caracteres

Además de los flujos basados en caracteres, Java proporciona las clases abstractas `Reader` y `Writer`, que son flujos basados en caracteres Unicode de dos bytes. La mayoría de los flujos basados en caracteres tienen sus correspondientes clases `Reader` o `Writer` basadas en caracteres.

Las clases `BufferedReader` (una subclase de la clase `abstract Reader`) y `BufferedWriter` (una subclase de la clase `abstract Writer`) permiten el uso del búfer para los flujos basados en caracteres. Recuerde que los flujos basados en caracteres utilizan caracteres Unicode; dichos flujos pueden procesar datos en cualquier lenguaje que sea representado por el conjunto de caracteres Unicode.

Las clases `CharArrayReader` y `CharArrayWriter` leen y escriben, respectivamente, un flujo de caracteres en un arreglo de caracteres. Un objeto `LineNumberReader` (una subclase de `BufferedReader`) es un flujo de caracteres con búfer que lleva el registro de los números de línea leídos (es decir, una nueva línea, un retorno o una combinación de retorno de carro y avance de línea). Puede ser útil llevar la cuenta de los números de línea si el programa necesita informar al lector sobre un error en una línea específica.

Las clases `FileReader` (una subclase de `InputStreamReader`) y `FileWriter` (una subclase de `OutputStreamWriter`) leen caracteres de, y escriben caracteres en, un archivo, respectivamente. Las clases `PipedReader` y `PipedWriter` implementan flujos de caracteres canalizados, que pueden utilizarse para transferir la información entre subprocesos. Las clases `StringReader` y `StringWriter` leen y escriben caracteres, respectivamente, en objetos `String`. Un objeto `PrintWriter` escribe caracteres en un flujo.

14.8 Abrir archivos con `JFileChooser`

La clase `JFileChooser` muestra un cuadro de diálogo (conocido como **cuadro de diálogo `JFileChooser`**) que permite al usuario seleccionar archivos o directorios con facilidad. Para demostrar este cuadro de diálogo, mejoramos el ejemplo de la sección 14.4, como se muestra en las figuras 14.22 y 14.23. El ejemplo ahora contiene una interfaz gráfica de usuario, pero sigue mostrando los mismos datos. El constructor llama al método `analizarRuta` en la línea 34. Después, este método llama al método `obtenerArchivo` en la línea 68 para obtener el objeto `File`.

El método `getFile` se define en las líneas 38 a 62 de la figura 14.22. En la línea 41 se crea un objeto `JFileChooser` y se asigna su referencia a `selectorArchivos`. En las líneas 42 y 43 se hace una llamada al método **`setFileSelectionMode`** para especificar lo que el usuario puede seleccionar del objeto `selectorArchivos`. Para este programa, utilizamos la constante `static` **`FILES_AND_DIRECTORIES`** de `JFileChooser` para indicar que pueden seleccionarse archivos y directorios. Otras constantes `static` son **`FILES_ONLY`** (sólo archivos y **`DIRECTORIES_ONLY`** (sólo directorios).

En la línea 45 se hace una llamada al método **`showOpenDialog`** para mostrar el cuadro de diálogo `JFileChooser` llamado **Abrir**. El argumento `this` especifica la ventana padre del cuadro de diálogo `JFileChooser`, la

```
// Fig. 14.22: DemostracionFile.java
// Demostración de la clase File.
import java.awt.BorderLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.io.File;
import javax.swing.JFileChooser;
import javax.swing.JFrame;
import javax.swing.JOptionPane;
import javax.swing.JScrollPane;
import javax.swing.JTextArea;
import javax.swing.JTextField;

public class DemostracionFile extends JFrame
{
   private JTextArea areaSalida; // se utiliza para salida
   private JScrollPane panelDespl; // se utiliza para que la salida pueda desplazarse

   // establece la GUI
   public DemostracionFile()
   {
      super( "Prueba de la clase File" );

      areaSalida = new JTextArea();

      // agrega areaSalida a panelDespl
      panelDespl = new JScrollPane( areaSalida );

      add( panelDespl, BorderLayout.CENTER ); // agrega panelDespl a la GUI

      setSize( 400, 400 ); // establece el tamaño de la GUI
      setVisible( true ); // muestra la GUI

      analizarRuta(); // crea y analiza un objeto File
   } // fin del constructor de DemostracionFile

   // permite al usuario especificar el nombre del archivo
   private File obtenerArchivo()
   {
      // muestra el cuadro de diálogo de archivos, para que el usuario pueda elegir el
      archivo a abrir
      JFileChooser selectorArchivos = new JFileChooser();
      selectorArchivos.setFileSelectionMode(
         JFileChooser.FILES_AND_DIRECTORIES );

      int resultado = selectorArchivos.showOpenDialog( this );

      // si el usuario hizo clic en el botón Cancelar en el cuadro de diálogo, regresa
      if ( resultado == JFileChooser.CANCEL_OPTION )
         System.exit( 1 );

      File nombreArchivo = selectorArchivos.getSelectedFile(); // obtiene el archivo
      seleccionado

      // muestra error si es inválido
      if ( ( nombreArchivo == null ) || ( nombreArchivo.getName().equals( "" ) ) )
      {
         JOptionPane.showMessageDialog( this, "Nombre de archivo inválido",
            "Nombre de archivo inválido", JOptionPane.ERROR_MESSAGE );
```

Figura 14.22 | Demostración de JFileChooser. (Parte 1 de 2).

```
58             System.exit( 1 );
59          } // fin de if
60
61          return nombreArchivo;
62       } // fin del método obtenerArchivo
63
64       // muestra información acerca del archivo que especifica el usuario
65       public void analizarRuta()
66       {
67          // crea un objeto File basado en la entrada del usuario
68          File nombre = obtenerArchivo();
69
70          if ( nombre.exists() ) // si el nombre existe, muestra información sobre él
71          {
72             // muestra la información sobre el archivo (o directorio)
73             areaSalida.setText( String.format(
74                "%s%s\n%s\n%s\n%s\n%s%s\n%s%s\n%s%s\n%s%s\n%s%s",
75                nombre.getName(), " existe",
76                ( nombre.isFile() ? "es un archivo" : "no es un archivo" ),
77                ( nombre.isDirectory() ? "es un directorio" :
78                   "no es un directorio" ),
79                ( nombre.isAbsolute() ? "es una ruta absoluta" :
80                   "no es una ruta absoluta" ), "Ultima modificacion: ",
81                nombre.lastModified(), "Tamanio: ", nombre.length(),
82                "Ruta: ", nombre.getPath(), "Ruta absoluta: ",
83                nombre.getAbsolutePath(), "Padre: ", nombre.getParent() ) );
84
85             if ( nombre.isDirectory() ) // imprime el listado del directorio
86             {
87                String directorio[] = nombre.list();
88                areaSalida.append( "\n\nContenido del directorio:\n" );
89
90                for ( String nombreDirectorio : directorio )
91                   areaSalida.append( nombreDirectorio + "\n" );
92             } // fin de else
93          } // fin de if exterior
94          else // no es archivo ni directorio, imprime mensaje de error
95          {
96             JOptionPane.showMessageDialog( this, nombre +
97                " no existe.", "ERROR", JOptionPane.ERROR_MESSAGE );
98          } // fin de else
99       } // fin del método analizarRuta
100   } // fin de la clase DemostracionFile
```

Figura 14.22 | Demostración de `JFileChooser`. (Parte 2 de 2).

cual determina la posición del cuadro de diálogo en la pantalla. Si se pasa `null`, el cuadro de diálogo se muestra en el centro de la pantalla; en caso contrario, el cuadro de diálogo se centra sobre la ventana de la aplicación (lo cual se especifica mediante el argumento `this`). Un cuadro de diálogo `JFileChooser` es un cuadro de diálogo modal que no permite al usuario interactuar con cualquier otra ventana en el programa, sino hasta que el usuario cierre el objeto `JFileChooser`, haciendo clic en el botón **Abrir** o **Cancelar**. El usuario selecciona la unidad, el nombre del directorio o archivo, y después hace clic en **Abrir**. El método `showOpenDialog` devuelve un entero, el cual especifica qué botón (**Abrir** o **Cancelar**) oprimió el usuario para cerrar el cuadro de diálogo. En la línea 48 se evalúa si el usuario hizo clic en **Cancelar**, para lo cual se compara el resultado con la constante `static` **`CANCEL_OPTION`**. Si son iguales, el programa termina. En la línea 51 se obtiene el archivo que seleccionó el usuario, llamando al método **`getSelectedFile`** de `selectorArchivos`. Después, el programa muestra información acerca del archivo o directorio seleccionado.

```java
// Fig. 14.23: PruebaDemostracionFile.java
// Prueba de la clase DemostracionFile.
import javax.swing.JFrame;

public class PruebaDemostracionFile
{
   public static void main( String args[] )
   {
      DemostracionFile aplicacion = new DemostracionFile();
      aplicacion.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
   } // fin de main
} // fin de la clase PruebaDemostracionFile
```

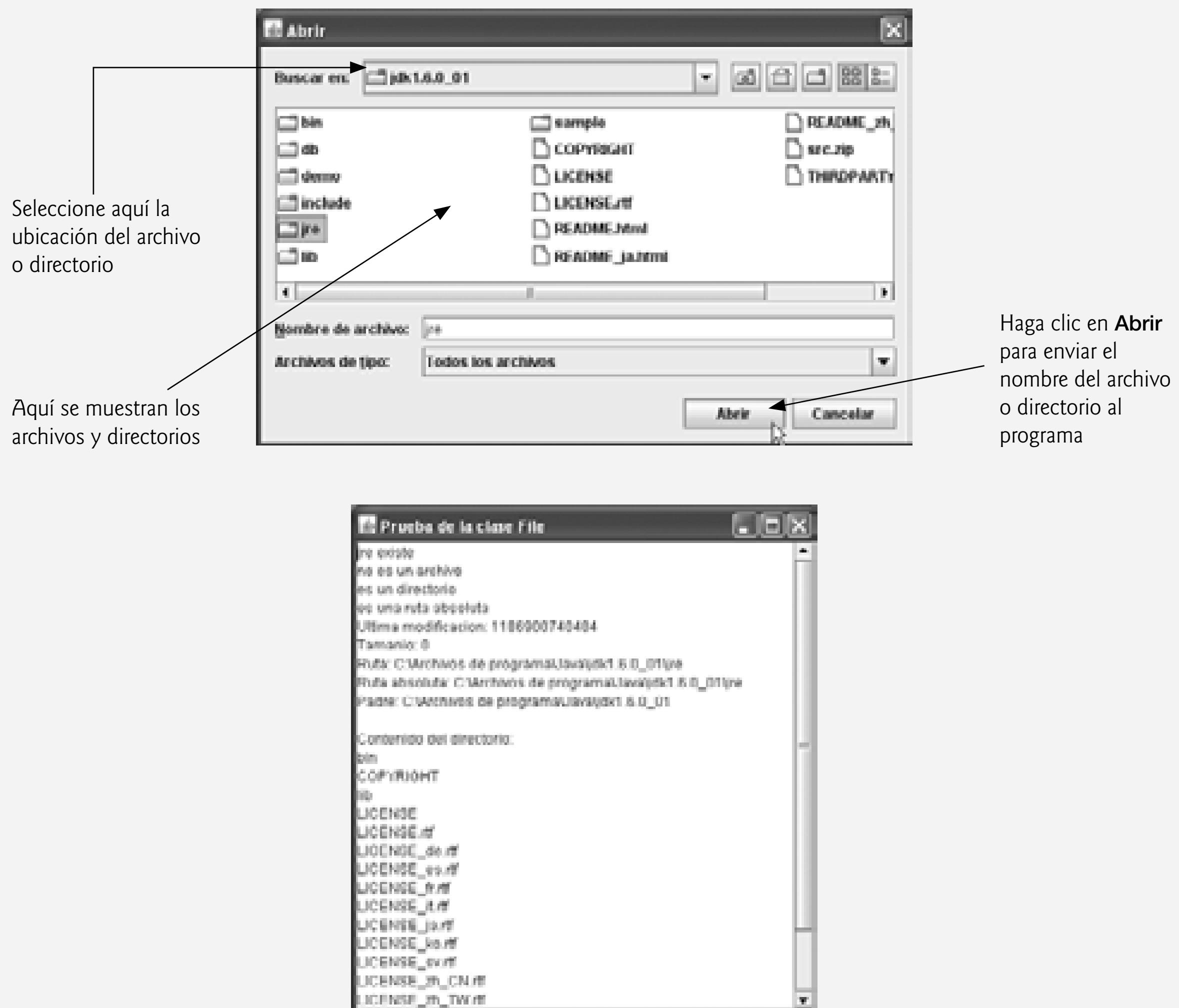


Figura 14.23 | Prueba de la clase `DemostracionFile`.

14.9 Conclusión

En este capítulo aprendió a utilizar el procesamiento de archivos para manipular datos persistentes. Aprendió que los datos se almacenan en las computadoras en forma de `0`s y `1`s, y que las combinaciones de estos valores se

utilizan para formar bytes, campos, registros y, en un momento dado, archivos. Comparamos los flujos basados en caracteres y los flujos basados en bytes, y presentamos varias clases para procesamiento de archivos que proporciona el paquete `java.io`. Utilizó la clase `File` para obtener información acerca de un archivo o directorio. Utilizó el procesamiento de archivos de acceso secuencial para manipular registros que se almacenan en orden, en base al campo clave del registro. Conoció las diferencias entre el procesamiento de archivos de texto y la serialización de objetos, y utilizó la serialización para almacenar y obtener objetos completos. El capítulo concluyó con una descripción general de las demás clases que proporciona el paquete `java.io`, y un pequeño ejemplo acerca del uso de un cuadro de diálogo `JFileChooser` para permitir a los usuarios seleccionar archivos de una GUI con facilidad. En el siguiente capítulo veremos el concepto de la recursividad: métodos que se llaman a sí mismos. Al definir métodos de esta forma, podemos producir programas más intuitivos.

Resumen

Sección 14.1 Introducción

- Los datos que se almacenan en variables y arreglos son temporales; se pierden cuando una variable local queda fuera de alcance, o cuando el programa termina. Las computadoras utilizan archivos para la retención a largo plazo de grandes cantidades de datos, incluso después de que los programas que crearon los datos terminan de ejecutarse.
- Los datos persistentes que se mantienen en archivos existen más allá de la duración de la ejecución del programa.
- Las computadoras almacenan los archivos en dispositivos de almacenamiento secundario, como los discos duros.

Sección 14.2 Jerarquía de datos

- El elemento de datos más pequeño en una computadora puede asumir el valor `0` o `1`, y se le conoce como bit. En última instancia, una computadora procesa todos los elementos de datos como combinaciones de ceros y unos.
- El conjunto de caracteres de la computadora es el conjunto de todos los caracteres que se utilizan para escribir programas y representar datos.
- Los caracteres en Java son Unicode y están compuestos de dos bytes, cada uno de los cuales se compone de ocho bits.
- Así como los caracteres están compuestos de bits, los campos se componen de caracteres o bytes. Un campo es un grupo de caracteres o bytes que transmite un significado.
- Los elementos de datos procesados por las computadoras forman una jerarquía de datos, la cual se vuelve más grande y compleja en estructura, a medida que progresamos de bits a caracteres, luego a campos, y así en lo sucesivo.
- Por lo general, varios campos componen un registro (que se implementa como `class` en Java).
- Un registro es un grupo de campos relacionados.
- Un archivo es un grupo de registros relacionados.
- Para facilitar la obtención de registros específicos de un archivo, se elije por lo menos un campo en cada registro como clave. Una clave de registro identifica que un registro pertenece a una persona o entidad específica, y es único para cada registro.
- Existen muchas formas de organizar los registros en un archivo. La más común se llama archivo secuencial, en el cual los registros se almacenan en orden, en base al campo clave de registro.
- Por lo general, a un grupo de archivos relacionados se le denomina base de datos. Una colección de programas diseñados para crear y administrar bases de datos se conoce como sistema de administración de bases de datos (DBMS).

Sección 14.3 Archivos y flujos

- Java ve a cada archivo como un flujo secuencial de bytes.
- Cada sistema operativo cuenta con un mecanismo para determinar el fin de un archivo, como un marcador de fin de archivo o la cuenta de los bytes totales en el archivo, que se registra en una estructura de datos administrativa, manejada por el sistema.
- Los flujos basados en bytes representan datos en formato binario.
- Los flujos basados en caracteres representan datos como secuencias de caracteres.
- Los archivos que se crean usando flujos basados en bytes son archivos binarios. Los archivos que se crean usando flujos basados en caracteres son archivos de texto. Los archivos de texto se pueden leer mediante editores de texto,

mientras que los archivos binarios se leen mediante un programa que convierte esos datos en un formato legible para los humanos.
- Java también puede asociar los flujos con distintos dispositivos. Tres objetos flujo se asocian con dispositivos cuando un programa de Java empieza a ejecutarse: `System.in`, `System.out` y `System.err`.
- El paquete `java.io` incluye definiciones para las clases de flujos, como `FileInputStream` (para la entrada basada en bytes de un archivo), `FileOutputStream` (para la salida basada en bytes hacia un archivo), `FileReader` (para la entrada basada en caracteres de un archivo) y `FileWriter` (para la salida basada en caracteres hacia un archivo). Los archivos se abren creando objetos de estas clases de flujos.

Sección 14.4 La clase `File`
- La clase `File` se utiliza para obtener información acerca de los archivos y directorios.
- Las operaciones de entrada y salida basadas en caracteres se pueden llevar a cabo con las clases `Scanner` y `Formatter`.
- La clase `Formatter` permite mostrar datos con formato en la pantalla, o enviarlos a un archivo, de una manera similar a `System.out.printf`.
- La ruta de un archivo o directorio especifica su ubicación en el disco.
- Una ruta absoluta contiene todos los directorios, empezando con el directorio raíz, que conducen hacia un archivo o directorio específico. Cada archivo o directorio en una unidad de disco tiene el mismo directorio raíz en su ruta.
- Por lo general, una ruta relativa empieza desde el directorio en el que se empezó a ejecutar la aplicación.
- Un carácter separador se utiliza para separar directorios y archivos en la ruta.

Sección 14.5 Archivos de texto de acceso secuencial
- Java no impone una estructura en un archivo; las nociones como los registros no existen como parte del lenguaje de Java. El programador debe estructurar los archivos para satisfacer los requerimientos de una aplicación.
- Para obtener datos de un archivo en forma secuencial, los programas comúnmente empiezan a leer desde el principio del archivo y leen todos los datos en forma consecutiva, hasta encontrar la información deseada.
- Los datos en muchos archivos secuenciales no se pueden modificar sin el riesgo de destruir otros datos en el archivo. Por lo tanto, los registros en un archivo de acceso secuencial normalmente no se actualizan directamente en su ubicación. En vez de ello, se vuelve a escribir el archivo completo.

Sección 14.6 Serialización de objetos
- Java cuenta con un mecanismo llamado serialización de objetos, el cual permite escribir o leer objetos completos mediante un flujo.
- Un objeto serializado es un objeto que se representa como una secuencia de bytes, e incluye los datos del objeto, así como información acerca del tipo del objeto y los tipos de datos almacenados en el mismo.
- Una vez que se escribe un objeto serializado en un archivo, se puede leer del archivo y deserializarse; es decir, se puede utilizar la información de tipo y los bytes que representan al objeto para recrearlo en la memoria.
- Las clases `ObjectInputStream` y `ObjectOutputStream`, que implementan en forma respectiva a las interfaces `ObjectInput` y `ObjectOutput`, permiten leer o escribir objetos completos de/a un flujo (posiblemente un archivo).
- Sólo las clases que implementan a la interfaz `Serializable` pueden serializarse y deserializarse con objetos `ObjectOutputStream` y `ObjectInputStream`.

Sección 14.7 Clases adicionales de `java.io`
- La interfaz `ObjectOutput` contiene el método `writeObject`, el cual recibe un objeto `Object` que implementa a la interfaz `Serializable` como argumento y escribe su información en un objeto `OutputStream`.
- La interfaz `ObjectInput` contiene el método `readObject`, que lee y devuelve una referencia a un objeto `Object` de un objeto `InputStream`. Una vez que se ha leído un objeto, su referencia puede convertirse al tipo actual del objeto.
- El uso de búfer es una técnica para mejorar el rendimiento de E/S. Con un objeto `BufferedOutputStream`, cada instrucción de salida no necesariamente produce una transferencia física real de datos al dispositivo de salida. En vez de ello, cada operación de salida se dirige hacia una región en memoria llamada búfer, la cual es lo bastante grande como para contener los datos de muchas operaciones de salida. La transferencia actual al dispositivo de salida se realiza entonces en una sola operación de salida física extensa cada vez que se llena el búfer.
- Con un objeto `BufferedInputStream`, muchos trozos "lógicos" de datos de un archivo se leen como una sola operación de entrada física extensa y se colocan en un búfer de memoria. A medida que un programa solicita cada nuevo trozo de datos, se obtiene del búfer. Cuando el búfer está vacío, se lleva a cabo la siguiente operación de entrada física real desde el dispositivo de entrada, para leer el nuevo grupo de trozos "lógicos" de datos.

Sección 14.8 Abrir archivos con ***JFileChooser***
- La clase JFileChooser se utiliza para mostrar un cuadro de diálogo, que permite a los usuarios de un programa seleccionar archivos con facilidad, mediante una GUI.

Terminología

aplicación de acceso directo
apuntador de posición de archivo
archivo
archivo binario
archivo de acceso secuencial
archivo de procesamiento por lotes
archivo de sólo lectura
archivo de texto
archivos de acceso directo
arreglo de bytes envuelto
base de datos
bit (dígito binario)
búfer
búfer de memoria
byte, tipo de datos
campo
CANCEL_OPTION, constante de la clase JFileChooser
canRead, método de la clase File
canWrite, método de la clase File
capacidad
-classpath, argumento de línea de comandos para java
-classpath, argumento de línea de comandos para javac
clave de registro
conjunto de caracteres
conjunto de caracteres ASCII (Código estándar estadounidense para el intercambio de información)
DataInput, interfaz
DataInputStream, clase
DataOutput, interfaz
DataOutputStream, clase
datos persistentes
dígito decimal
DIRECTORIES_ONLY, constante de la clase JFileChooser
directorio
directorio padre
directorio raíz
disco
disco óptico
dispositivos de almacenamiento secundario
EndOfFileException, excepción
envoltura de objetos flujo
exists, método de la clase File
exit, método de la clase System
File, clase
FileInputStream, clase
FileOutputStream, clase
FileReader, clase
FILES_AND_DIRECTORIES, constante de la clase JFileChooser
FILES_ONLY, constante de la clase JFileChooser
FileWriter, clase
flujo basado en bytes
flujo basado en caracteres
flujo de bytes
Formatter, clase
getAbsolutePath, método de la clase File
getName, método de la clase File
getParent, método de la clase File
getPath, método de la clase File
getSelectedFile, método de la clase JFileChooser
InputStream, clase
interfaz de marcado
IOException, excepción
isAbsolute, método de la clase File
isDirectory, método de la clase File
isFile, método de la clase File
java.io, paquete
jerarquía de datos
JFileChooser, clase
JFileChooser, cuadro de diálogo
lastModified, método de la clase File
length, método de la clase File
list, método de la clase File
marcador de fin de archivo
nombre de directorio
NoSuchElementException, excepción
ObjectInputStream, clase
ObjectOutputStream, clase
objeto deserializado
objeto flujo
objeto flujo de error estándar
objeto serializado
operación física de entrada
operación física de salida
operaciones lógicas de entrada
operaciones lógicas de salida
OutputStream, clase
pathSeparator, campo static de la clase File
PrintStream, clase
PrintWriter, clase
procesamiento de archivos
procesamiento de flujos
Reader, clase
readLine, método de la clase BufferedReader
readObject, método de la clase ObjectInputStream
readObject, método de la interfaz ObjectInput
registro
registro de longitud fija
ruta absoluta

ruta relativa
secuencia de comandos de shell
Serializable, interfaz
serialización de objetos
setErr, método de la clase System
setIn, método de la clase System
setOut, método de la clase System
setSelectionMode de la clase JFileChooser
showOpenDialog de la clase JFileChooser
sistema de administración de bases de datos (DBMS)
System.err (flujo de error estándar)
transient, palabra clave
truncada
Unicode, conjunto de caracteres
URI (Identificador uniforme de recursos)
writeBoolean, método de la interfaz DataOutput
writeByte, método de la interfaz DataOutput
writeBytes, método de la interfaz DataOutput
writeChar, método de la interfaz DataOutput
writeChars, método de la interfaz DataOutput
writeDouble, método de la interfaz DataOutput
writeFloat, método de la interfaz DataOutput
writeInt, método de la interfaz DataOutput
writeLong, método de la interfaz DataOutput
writeObject, método de la clase ObjectOutputStream
writeObject, método de la interfaz ObjectOutput
Writer, clase
writeShort, método de la interfaz DataOutput
writeUTF, método de la interfaz DataOutput

Ejercicios de autoevaluación

14.1 Complete las siguientes oraciones:

a) Básicamente, todos los elementos de datos procesados por una computadora se reducen en combinaciones de ______________ y ______________.
b) El elemento de datos más pequeño que puede procesar una computadora se conoce como ____________.
c) Un ____________ puede verse algunas veces como un grupo de registros relacionados.
d) Los dígitos, letras y símbolos especiales se conocen como____________.
e) Una base de datos es un grupo de ____________ relacionados.
f) El objeto ____________ normalmente permite a un programa imprimir mensajes de error en la pantalla.

14.2 Conteste con *verdadero* o *falso* a cada una de las siguientes proposiciones; en caso de ser *falso*, explique por qué.

a) El programador debe crear explícitamente los objetos flujo System.in, System.out y System.err.
b) Al leer datos de un archivo mediante la clase Scanner, si el programador desea leer datos en el archivo varias veces, el archivo debe cerrarse y volver a abrirse para leer desde el principio del archivo. Esto desplaza el apuntador de posición de archivo de vuelta hasta el principio del archivo.
c) El método exists de la clase File devuelve true si el nombre que se especifica como argumento para el constructor de File es un archivo o directorio en la ruta especificada.
d) Los archivos binarios pueden ser leídos por los humanos.
e) Una ruta absoluta contiene todos los directorios, empezando con el directorio raíz, que conducen hacia un archivo o directorio específico.
f) La clase Formatter contiene el método printf, que permite imprimir datos con formato en la pantalla, o enviarlos a un archivo.

14.3 Complete las siguientes tareas; suponga que cada una se aplica al mismo programa:

a) Escriba una instrucción que abra el archivo "antmaest.txt" en modo de entrada; use la variable Scanner llamada entAntMaestro.
b) Escriba una instrucción que abra el archivo "trans.txt" en modo de entrada; use la variable Scanner llamada entTransaccion.
c) Escriba una instrucción para abrir el archivo "nuevomaest.txt" en modo de salida (y creación); use la variable Formatter llamada salNuevoMaest.
d) Escriba las instrucciones necesarias para leer un registro del archivo "antmaest.txt". Los datos leídos deben usarse para crear un objeto de la clase RegistroCuenta; use la variable Scanner llamada entAntMaest. Suponga que la clase RegistroCuenta es la misma que la de la figura 14.6.
e) Escriba las instrucciones necesarias para leer un registro del archivo "trans.txt". El registro es un objeto de la clase RegistroTransaccion; use la variable Scanner llamada entTransaccion. Suponga que la clase RegistroTransaccion contiene el método establecerCuenta (que recibe un int) para establecer el número de cuenta, y el método establecerMonto (que recibe un double) para establecer el monto de la transacción.

f) Escriba una instrucción que escriba un registro en el archivo "nuevomaest.txt". El registro es un objeto de tipo RegistroCuenta; use la variable Formatter llamada salNuevoMaest.

14.4 Complete las siguientes tareas, suponiendo que cada una se aplica al mismo programa:

a) Escriba una instrucción que abra el archivo "antmaest.ser" en modo de entrada; use la variable ObjectInputStream llamada entAntMaest para envolver un objeto FileInputStream.

b) Escriba una instrucción que abra el archivo "trans.ser" en modo de entrada; use la variable ObjectInputStream llamada entTransaccion para envolver un objeto FileInputStream.

c) Escriba una instrucción para abrir el archivo "nuevomaest.ser" en modo de salida (y creación); use la variable ObjectOutputStream llamada salNuevoMaest para envolver un objeto FileOutputStream.

d) Escriba una instrucción que lea un registro del archivo "antmaest.ser". El registro es un objeto de la clase RegistroCuentaSerializable; use la variable ObjectInputStream llamada entAntMaestro. Suponga que la clase RegistroCuentaSerializable es igual que la de la figura 14.17.

e) Escriba una instrucción que lea un registro del archivo "trans.ser". El registro es un objeto de la clase RegistroTransaccion; use la variable ObjectInputStream llamada entTransaccion.

f) Escriba una instrucción que escriba un registro en el archivo "nuevomaest.ser". El registro es un objeto de tipo RegistroCuenta; use la variable Formatter llamada salNuevoMaest.

14.5 Encuentre el error en cada uno de los siguientes bloques de código y muestre cómo corregirlo.

a) Suponga que se declaran cuenta, compania y monto.

```
ObjectOutputStream flujoSalida;

flujoSalida.writeInt( cuenta );
flujoSalida.writeChars( compania );
flujoSalida.writeDouble( monto );
```

b) Las siguientes instrucciones deben leer un registro del archivo "porpagar.txt". Se debe utilizar la variable entPorPagar de Scanner para hacer referencia a este archivo.

```
Scanner entPorPagar = new Scanner( new File( "porpagar.txt") );
RegistroPorPagar registro = ( RegistroPorPagar ) entPorPagar.readObject();
```

Respuestas a los ejercicios de autoevaluación

14.1 a) unos, ceros. b) bit. c) archivo. d) caracteres. e) archivos. f) System.err.

14.2 a) Falso. Estos tres flujos se crean para el programador cuando se empieza a ejecutar una aplicación de Java.

b) Verdadero.

c) Verdadero.

d) Falso. Los archivos de texto pueden ser leídos por los humanos.

e) Verdadero.

f) Falso. La clase Formatter contiene el método format, el cual permite imprimir datos con formato en la pantalla, o enviarlos a un archivo.

14.3

a)
```
Scanner entAntMaest = new Scanner( new File( "antmaest.txt" ) );
```
b)
```
Scanner entTransaccion = new Scanner( new File( "trans.txt" ) );
```
c)
```
Formatter salNuevoMaest = new Formatter( "nuevomaest.txt" );
```
d)
```
RegistroCuenta  cuenta = new RegistroCuenta();
    cuenta.establecerCuenta( entAntMaest.nextInt() );
    cuenta.establecerPrimerNombre( entAntMaest.next() );
    cuenta.establecerApellidoPaterno( entAntMaest.next() );
    cuenta.establecerSaldo( entAntMaest.nextDouble() );
```
e)
```
RegistroTransaccion transacción = new Transacción();
    transaccion.establecerCuenta( entTransaccion.nextInt() );
    transaccion.establecerMonto( entTransaccion.nextDouble() );
```
f)
```
salNuevMaest.format( "%d %s %s &.2f\n",
    cuenta.obtenerCuenta(), cuenta.obtenerPrimerNombre(),
    cuenta.obtenerApellidoPaterno(), cuenta.obtenerSaldo() );
```

14.4 a)
```
ObjectInputStream entAntMaest = new ObjectInputStream(
    new FileInputStream( "antmaest.ser" ) );
```
b)
```
ObjectInputStream entTransaccion = new ObjectInputStream(
    new FileInputStream( "trans.ser" ) );
```
c)
```
ObjectOutputStream salNuevMaest = new ObjectOutputStream(
    new FileOutputStream( "nuevmaest.ser" ) );
```
d)
```
registroCuenta = ( RegistroCuentaSerializable ) entAntMaest.readObject();
```
e)
```
registroTransaccion = ( RegistroTransaccion ) entTransaccion.readObject();
```
f)
```
salNuevMaest.writeObject( nuevoRegistroCuenta );
```

14.5 a) Error: el archivo no se ha abierto antes de tratar de enviar datos al flujo.
Corrección: abrir un archivo en modo de salida, creando un nuevo objeto `ObjectOutputStream` que envuelva a un objeto `FileOutputStream`.

b) Error: este ejemplo utiliza archivos de texto con un objeto `Scanner`, no hay serialización de objetos. Como resultado, el método `readObject` no puede usarse para leer esos datos del archivo. Cada pieza de datos debe leerse por separado y después utilizarse para crear un objeto `RegistroPorPagar`.
Corrección: utilice los métodos de `entPorPagar` para leer cada pieza del objeto `RegistroPorPagar`.

Ejercicios

14.6 Llene los espacios en blanco en cada uno de los siguientes enunciados:

a) Las computadoras almacenan grandes cantidades de datos en dispositivos de almacenamiento secundario, como ____________.
b) Un ____________ está compuesto de varios campos.
c) Para facilitar la recuperación de registros específicos de un archivo, debe seleccionarse un campo en cada registro como ____________.
d) Los archivos que se crean usando flujos basados en bytes se conocen como archivos ____________, mientras que los archivos creados usando flujos basados en caracteres se conocen como archivos ____________.
e) Los objetos flujo estándar son ____________, ____________ y ____________.

14.7 Determine cuál de los siguientes enunciados es *verdadero* y cuál es *falso*. Si es *falso*, explique por qué.

a) Las impresionantes funciones realizadas por las computadoras involucran esencialmente la manipulación de ceros y unos.
b) Las personas especifican los programas y elementos de datos como caracteres. Después, las computadoras manipulan y procesan estos caracteres como grupos de ceros y unos.
c) Los elementos de datos que se representan en las computadoras forman una jerarquía de datos, en la cual los elementos de datos se hacen más grandes y complejos, a medida que progresamos de campos a caracteres, de caracteres a bits, etcétera.
d) Una clave de registro identifica que un registro pertenece a un campo específico.
e) Las compañías almacenan toda su información en un solo archivo, para poder facilitar el procesamiento computacional de la información. Cuando un programa crea un archivo, éste es retenido automáticamente por la computadora para cuando se haga referencia a él en un futuro.

14.8 *(Asociación de archivos)* El ejercicio de autoevaluación 14.3 pide al lector que escriba una serie de instrucciones individuales. En realidad, estas instrucciones forman el núcleo de un tipo importante de programa para procesar archivos: un programa para asociar archivos. En el procesamiento de datos comercial, es común tener varios archivos en cada sistema de aplicaciones. Por ejemplo, en un sistema de cuentas por cobrar hay generalmente un archivo maestro, el cual contiene información detallada acerca de cada cliente, como su nombre, dirección, número telefónico, saldo deudor, límite de crédito, términos de descuento, acuerdos contractuales y posiblemente un historial condensado de compras recientes y pagos en efectivo.

A medida que ocurren las transacciones (es decir, a medida que se generan las ventas y llegan los pagos en el correo), la información acerca de ellas se introduce en un archivo. Al final de cada periodo de negocios (un mes para algunas compañías, una semana para otras y un día en algunos casos), el archivo de transacciones (llamado `"trans.txt"`) se aplica al archivo maestro (llamado `"antmaest.txt"`) para actualizar el registro de compras y pagos de cada cuenta. Durante una actualización, el archivo maestro se rescribe como el archivo `"nuevomaest.txt"`, el cual se utiliza al final del siguiente periodo de negocios para empezar de nuevo el proceso de actualización.

Los programas para asociar archivos deben tratar con ciertos problemas que no existen en programas de un solo archivo. Por ejemplo, no siempre ocurre una asociación. Si un cliente en el archivo maestro no ha realizado compras ni pagos en efectivo en el periodo actual de negocios, no aparecerá ningún registro para este cliente en el archivo de transacciones. De manera similar, un cliente que haya realizado compras o pagos en efectivo podría haberse mudado recientemente a esta comunidad, y tal vez la compañía no haya tenido la oportunidad de crear un registro maestro para este cliente.

Escriba un programa completo para asociar archivos de cuentas por cobrar. Utilice el número de cuenta en cada archivo como la clave de registro para fines de asociar los archivos. Suponga que cada archivo es un archivo de texto secuencial con registros almacenados en orden ascendente, por número de cuenta.

a) Defina la clase `RegistroTransaccion`. Los objetos de esta clase contienen un número de cuenta y un monto para la transacción. Proporcione métodos para modificar y obtener estos valores.
b) Modifique la clase `RegistroCuenta` de la figura 14.6 para incluir el método `combinar`, el cual recibe un objeto `RegistroTransaccion` y combina el saldo del objeto `RegistroCuenta` con el valor del monto del objeto `RegistroTransaccion`.
c) Escriba un programa para crear datos de prueba para el programa. Use los datos de la cuenta de ejemplo de las figuras 14.24 y 14.25. Ejecute el programa para crear los archivos `trans.txt` y `antmaest.txt`, para que los utilice su programa de asociación de archivos.
d) Cree la clase `AsociarArchivos` para llevar a cabo la funcionalidad de asociación de archivos. La clase debe contener métodos para leer `antmaest.txt` y `trans.txt`. Cuando ocurra una coincidencia (es decir, que aparezcan registros con el mismo número de cuenta en el archivo maestro y en el archivo de transacciones), sume el monto en dólares del registro de transacciones al saldo actual en el registro maestro, y escriba el registro `"nuevomaest.txt"`. (Suponga que las compras se indican mediante montos positivos en el archivo de transacciones, y los pagos mediante montos negativos). Cuado haya un registro maestro para una cuenta específica, pero no haya un registro de transacciones correspondiente, simplemente escriba el registro maestro en `"nuevomaest.txt"`. Cuando haya un registro de transacciones pero no un registro maestro correspondiente, imprima en un archivo de registro el mensaje `"Hay un registro de transacciones no asociado para ese numero de cliente..."` (utilice el número de cuenta del registro de transacciones). El archivo de registro debe ser un archivo de texto llamado `"registro.txt"`.

14.9 *(Asociación de archivos con varias transacciones)* Es posible (y muy común) tener varios registros de transacciones con la misma clave de registro. Esta situación ocurre cuando un cliente hace varias compras y pagos en efectivo durante un periodo de negocios. Rescriba su programa para asociar archivos de cuentas por cobrar del ejercicio 14.8, para proporcionar la posibilidad de manejar varios registros de transacciones con la misma clave de registro. Modifique los datos de prueba de `CrearDatos.java` para incluir los registros de transacciones adicionales de la figura 14.26.

14.10 *(Asociación de archivos con serialización de objetos)* Vuelva a crear su solución para el ejercicio 14.9, usando la serialización de objetos. Use las instrucciones del ejercicio 14.4 como base para este programa. Tal vez sea conveniente crear aplicaciones para que lean los datos almacenados en los archivos `.ser`; puede modificar el código de la sección 14.6.2 para este fin.

Número de cuenta	Nombre	Saldo
100	Alan Jones	348.17
300	Mary Smith	27.19
500	Sam Sharp	0.00
700	Susy Green	-14.22

Figura 14.24 | Datos de ejemplo para el archivo maestro.

Archivo de transacciones Número de cuenta	Monto de la transacción
100	27.14
300	62.11
400	100.56
900	82.17

Figura 14.25 | Datos de ejemplo para el archivo de transacciones.

Número de cuenta	Monto en dólares
300	83.89
700	80.78
700	1.53

Figura 14.26 | Registros de transacciones adicionales.

14.11 *(Generador de palabras de números telefónicos)* Los teclados telefónicos estándar contienen los dígitos del cero al nueve. Cada uno de los números del dos al nueve tiene tres letras asociadas (figura 14.27). A muchas personas se les dificulta memorizar números telefónicos, por lo que utilizan la correspondencia entre los dígitos y las letras para desarrollar palabras de siete letras que corresponden a sus números telefónicos. Por ejemplo, una persona cuyo número telefónico sea 686-3767 podría utilizar la correspondencia indicada en la figura 14.27 para desarrollar la palabra de siete letras "NUMEROS". Cada palabra de siete letras corresponde exactamente a un número telefónico de siete dígitos. El restaurante que desea incrementar su negocio de comidas para llevar podría lograrlo utilizando el número 266-4327 (es decir, "COMIDAS").

Dígito	Letras
2	A B C
3	D E F
4	G H I
5	J K L
6	M N O
7	P R S
8	T U V
9	W X Y

Figura 14.27 | Dígitos y letras de los teclados telefónicos.

Cada número telefónico de siete letras corresponde a muchas palabras de siete letras distintas. Desafortunadamente, la mayoría de estas palabras representan yuxtaposiciones irreconocibles de letras. Sin embargo, es posible que el dueño de una carpintería se complazca en saber que el número telefónico de su taller, 683-2537, corresponde a

"MUEBLES". El propietario de una tienda de licores estaría, sin duda, feliz de averiguar que el número telefónico 232-4327 corresponde a "BEBIDAS". Un veterinario con el número telefónico 627-2682 se complacería en saber que ese número corresponde a las letras "MASCOTA". El propietario de una tienda de música estaría complacido en saber que su número telefónico 687-4225 corresponde a "MUSICAL".

Escriba un programa que, dado un número de siete dígitos, utilice un objeto `PrintStream` para escribir en un archivo todas las combinaciones posibles de palabras de siete letras que corresponden a ese número. Hay 2,187 (3^7) combinaciones posibles. Evite los números telefónicos con los dígitos 0 y 1.

14.12 *(Encuesta estudiantil)* La figura 7.8 contiene un arreglo de respuestas a una encuesta, el cual está codificado directamente en el programa. Suponga que deseamos procesar resultados de encuestas que se guarden en un archivo. Este ejercicio requiere de dos programas separados. Primero, cree una aplicación que pida al usuario las respuestas de la encuesta y que escriba cada respuesta en un archivo. Utilice un objeto `Formatter` para crear un archivo llamado `numeros.txt`. Cada entero debe escribirse utilizando el método `format`. Después modifique el programa de la figura 7.8 para leer las respuestas de la encuesta del archivo `numeros.txt`. Las respuestas deben leerse del archivo mediante el uso de un objeto `Scanner`. Deberá utilizar el método `nextInt` para introducir un entero del archivo a la vez. El programa deberá seguir leyendo respuestas hasta que llegue al fin del archivo. Los resultados deberán escribirse en el archivo de texto `"salida.txt"`.

14.13 Modifique el ejercicio 11.18 para permitir que el usuario guarde un dibujo en un archivo, o cargue un dibujo anterior de un archivo, usando la serialización de objetos. Agregue los botones **Cargar** (para leer objetos de un archivo), **Guardar** (para escribir objetos en un archivo) y **Generar figuras** (para mostrar un conjunto aleatorio de figuras en la pantalla). Use un objeto `ObjectOutputStream` para escribir en el archivo y un objeto `ObjectInputStream` para leer del archivo. Escriba el arreglo de objetos `MiFigura` usando el método `writeObject` (clase `ObjectOutputStream`) y lea el arreglo usando el método `readObject` (`ObjectInputStream`). Observe que el mecanismo de serialización de objetos puede leer o escribir arreglos completos; no es necesario manipular cada elemento del arreglo de objetos `MiFigura` por separado. Simplemente se requiere que todas las figuras sean `Serializable`. Para los botones **Cargar** y **Guardar**, use un objeto `JFileChooser` para permitir que el usuario seleccione el archivo en el que se almacenarán las figuras, o del que se leerán. Cuando el usuario ejecute el programa por primera vez, no se mostrarán figuras en la pantalla. El usuario puede mostrar figuras abriendo un archivo de figuras previamente guardado, o haciendo clic en el botón **Generar figuras**. Cuando el usuario haga clic en este botón, la aplicación deberá generar un número aleatorio de figuras, hasta un total de 15. Una vez que haya figuras en la pantalla, los usuarios podrán guardarlas en un archivo, usando el botón **Guardar**.

15

Recursividad



Debemos aprender a explorar todas las opciones y posibilidades a las que nos enfrentamos en un mundo complejo, que evoluciona rápidamente.

—James William Fulbright

Oh, maldita iteración, que eres capaz de corromper hasta a un santo.

—William Shakespeare

Es un pobre orden de memoria, que sólo funciona al revés.

—Lewis Carroll

La vida sólo puede comprenderse al revés; pero debe vivirse hacia delante.

—Soren Kierkegaard

Empujen; sigan avanzando.

—Thomas Morton

OBJETIVOS

En este capítulo aprenderá a:

- Comprender el concepto de recursividad.
- Escribir y utilizar métodos recursivos.
- Determinar el caso base y el paso de recursividad en un algoritmo recursivo.
- Conocer cómo el sistema maneja las llamadas a métodos recursivos.
- Conocer las diferencias entre recursividad e iteración, y cuándo es apropiado utilizar cada una.
- Conocer las figuras geométricas llamadas fractales, y cómo se dibujan mediante la recursividad.
- Conocer el concepto de "vuelta atrás" recursiva (backtracking), y por qué es una técnica efectiva para solucionar problemas.

Plan general

15.1 Introducción

Los programas que hemos visto hasta ahora están estructurados generalmente como métodos que se llaman entre sí, de una manera disciplinada y jerárquica. Sin embargo, para algunos problemas es conveniente hacer que un método se llame a sí mismo. Dicho método se conoce como **método recursivo**; este método se puede llamar en forma directa o indirecta a través de otro método. La recursividad es un tema importante, que puede tratarse de manera extensa en los cursos de ciencias computacionales de nivel superior. En este capítulo consideraremos la recursividad en forma conceptual, y después presentaremos varios programas que contienen métodos recursivos. En la figura 15.1 se sintetizan los ejemplos y ejercicios de recursividad que se incluyen en este libro.

Capítulo	Ejemplos y ejercicios de recursividad en este libro
15	Método factorial (figuras 15.3 y 15.4.) Método Fibonacci (figuras 15.5 y 15.6). Torres de Hanoi (figuras 15.13 y 15.14). Fractales (figuras 15.21 y 15.22). ¿Qué hace este código? (ejercicios 15.7, 15.12 y 15.13). Encuentre el error en el siguiente código (ejercicio 15.8). Elevar un entero a una potencia entera (ejercicio 15.9). Visualización de la recursividad (ejercicio 15.10). Máximo común divisor (ejercicio 15.11). Determinar si una cadena es un palíndromo (ejercicio 15.14). Ocho reinas (ejercicio 15.15). Imprimir un arreglo (ejercicio 15.16). Imprimir un arreglo al revés (ejercicio 15.17). Mínimo valor en un arreglo (ejercicio 15.18). Fractal de estrella (ejercicio 15.19). Recorrido de un laberinto mediante el uso de la "vuelta atrás" recursiva (ejercicio 15.20). Generación de laberintos al azar (ejercicio 15.21). Laberintos de cualquier tamaño (ejercicio 15.22). Tiempo para calcular números de Fibonacci (ejercicio 15.23).
16	Ordenamiento por combinación (figuras 16.10 y 16.11). Búsqueda lineal recursiva (ejercicio 16.8). Búsqueda binaria recursiva (ejercicio 16.9). Quicksort (ejercicio 16.10).

Figura 15.1 | Resumen de los ejemplos y ejercicios de recursividad en este libro. (Parte 1 de 2).

Capítulo	Ejemplos y ejercicios de recursividad en este libro
17	Inserción en árbol binario (figura 17.17). Recorrido preorden de un árbol binario (figura 17.17). Recorrido inorden de un árbol binario (figura 17.17). Recorrido postorden de un árbol binario (figura 17.17). Imprimir una lista en forma recursiva y en forma inversa (ejercicio 17.20). Buscar en una lista en forma recursiva (ejercicio 17.21).

Figura 15.1 | Resumen de los ejemplos y ejercicios de recursividad en este libro. (Parte 2 de 2).

15.2 Conceptos de recursividad

Los métodos para solucionar problemas recursivos tienen varios elementos en común. Cuando se hace una llamada a un método recursivo para resolver un problema, el método en realidad es capaz de resolver sólo el (los) caso(s) más simple(s), o **caso(s) base**. Si se hace la llamada al método con un caso base, el método devuelve un resultado. Si se hace la llamada al método con un problema más complejo, el método comúnmente divide el problema en dos piezas conceptuales: una pieza que el método sabe cómo resolver y otra pieza que no sabe cómo resolver. Para que la recursividad sea factible, esta última pieza debe ser similar al problema original, pero una versión ligeramente más sencilla o simple del mismo. Debido a que este nuevo problema se parece al problema original, el método llama a una nueva copia de sí mismo para trabajar en el problema más pequeño; a esto se le conoce como **llamada recursiva**, y también como **paso recursivo**. Por lo general, el paso recursivo incluye una instrucción `return`, ya que su resultado se combina con la parte del problema que el método supo cómo resolver, para formar un resultado que se pasará de vuelta al método original que hizo la llamada. Este concepto de separar el problema en dos porciones más pequeñas es una forma del método "divide y vencerás" que presentamos en el capítulo 6.

El paso recursivo se ejecuta mientras siga activa la llamada original al método (es decir, que no haya terminado su ejecución). Se pueden producir muchas llamadas recursivas más, a medida que el método divide cada nuevo subproblema en dos piezas conceptuales. Para que la recursividad termine en un momento dado, cada vez que el método se llama a sí mismo con una versión más simple del problema original, la secuencia de problemas cada vez más pequeños debe converger en un caso base. En ese punto, el método reconoce el caso base y devuelve un resultado a la copia anterior del método. Después se origina una secuencia de retornos, hasta que la llamada al método original devuelve el resultado final al método que lo llamó.

Un método recursivo puede llamar a otro método, que a su vez puede hacer una llamada de vuelta al método recursivo. A dicho proceso se le conoce como **llamada recursiva indirecta** o **recursividad indirecta**. Por ejemplo, el método `A` llama al método `B`, que hace una llamada de vuelta al método `A`. Esto se sigue considerando como recursividad, debido a que la segunda llamada al método `A` se realiza mientras la primera sigue activa; es decir, la primera llamada al método `A` no ha terminado todavía de ejecutarse (debido a que está esperando que el método `B` le devuelva un resultado) y no ha regresado al método original que llamó al método `A`.

Para comprender mejor el concepto de recursividad, veamos un ejemplo que es bastante común para los usuarios de computadora: la definición recursiva de un directorio en una computadora. Por lo general, una computadora almacena los archivos relacionados en un directorio. Este directorio puede estar vacío, puede contener archivos y/o puede contener otros directorios (que, por lo general, se conocen como subdirectorios). A su vez, cada uno de estos directorios puede contener también archivos y directorios. Si queremos enlistar cada archivo en un directorio (incluyendo todos los archivos en los subdirectorios de ese directorio), necesitamos crear un método que lea primero los archivos del directorio inicial y que después haga llamadas recursivas para enlistar los archivos en cada uno de los subdirectorios de ese directorio. El caso base ocurre cuando se llega a un directorio que no contenga subdirectorios. En este punto, se han enlistado todos los archivos en el directorio original y no se necesita más la recursividad.

15.3 Ejemplo de uso de recursividad: factoriales

Escribimos un programa recursivo, para realizar un popular cálculo matemático. Considere el factorial de un entero positivo n, escrito como $n!$ (y se pronuncia como "factorial de n"), que viene siendo el producto

$n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 1$

en donde 1! es igual a 1 y 0! se define como 1. Por ejemplo, 5! es el producto $5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$, que es igual a 120.

El factorial del entero `numero` (en donde `numero ≥ 0`) puede calcularse de manera **iterativa** (sin recursividad), usando una instrucción `for` de la siguiente manera:

```
factorial = 1;

for ( int contador = numero; contador >= 1; contador-- )
    factorial *= contador;
```

Podemos llegar a una declaración recursiva del método del factorial, si observamos la siguiente relación:

$n! = n \cdot (n-1)!$

Por ejemplo, 5! es sin duda igual a $5 \cdot 4!$, como se muestra en las siguientes ecuaciones:

$5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$
$5! = 5 \cdot (4 \cdot 3 \cdot 2 \cdot 1)$
$5! = 5 \cdot (4!)$

La evaluación de 5! procedería como se muestra en la figura 15.2. La figura 15.2(a) muestra cómo procede la sucesión de llamadas recursivas hasta que 1! (el caso base) se evalúa como 1, lo cual termina la recursividad. La figura 15.2(b) muestra los valores devueltos de cada llamada recursiva al método que hizo la llamada, hasta que se calcula y devuelve el valor final.

En la figura 15.3 se utiliza la recursividad para calcular e imprimir los factoriales de los enteros del 0 al 10. El método recursivo `factorial` (líneas 7 a 13) realiza primero una evaluación para determinar si una condición de terminación (línea 9) es `true`. Si `numero` es menor o igual que `1` (el caso base), `factorial` devuelve `1`, ya no es necesaria más recursividad y el método regresa. Si `numero` es mayor que `1`, en la línea 12 se expresa el problema como el producto de `numero` y una llamada recursiva a `factorial` en la que se evalúa el factorial de `numero - 1`, el cual es un problema un poco más pequeño que el cálculo original, `factorial( numero )`.

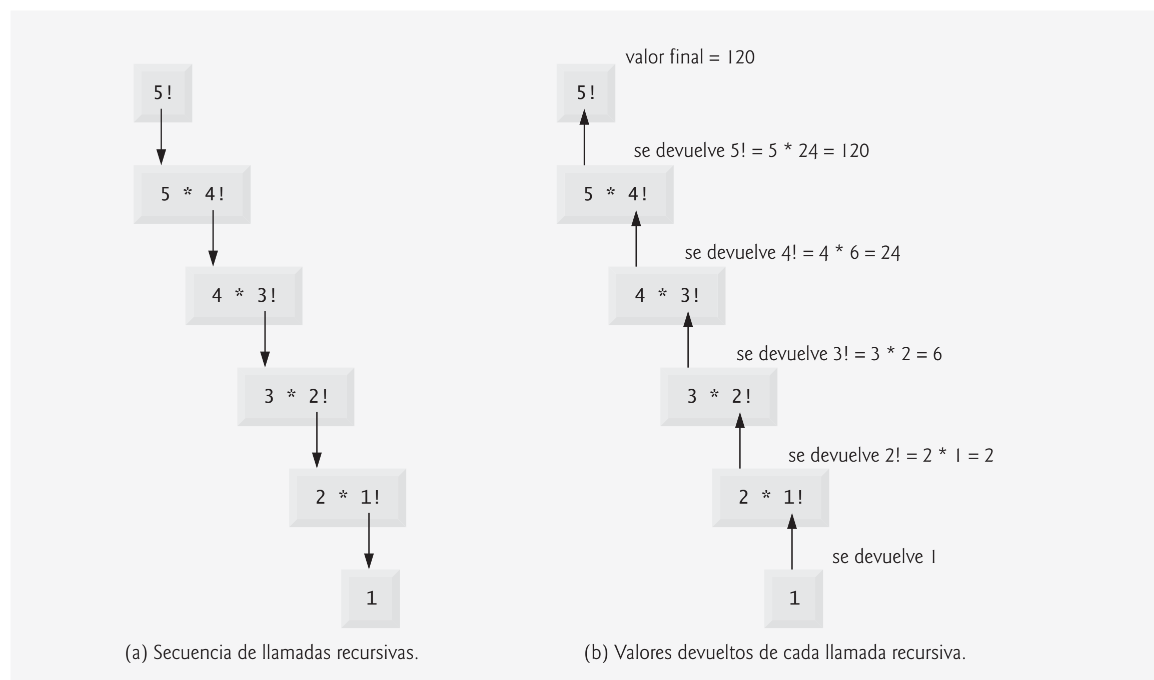


Figura 15.2 | Evaluación recursiva de 5!.

```
1  // Fig. 15.3: CalculoFactorial.java
2  // Método factorial recursivo.
3
4  public class CalculoFactorial
5  {
6     // declaración recursiva del método factorial
7     public long factorial( long numero )
8     {
9        if ( numero <= 1 ) // evalúa el caso base
10          return 1; // casos base: 0! = 1 y 1! = 1
11       else // paso recursivo
12          return numero * factorial( numero - 1 );
13    } // fin del método factorial
14
15    // imprime factoriales para los valores del 0 al 10
16    public void mostrarFactoriales()
17    {
18       // calcula los factoriales del 0 al 10
19       for ( int contador = 0; contador <= 10; contador++ )
20          System.out.printf( "%d! = %d\n", contador, factorial( contador ) );
21    } // fin del método mostrarFactoriales
22 } // fin de la clase CalculoFactorial
```

Figura 15.3 | Cálculos de factoriales con un método recursivo.



Error común de programación 15.1

Si omitimos el caso base o escribimos el paso recursivo en forma incorrecta, de manera que no converja en el caso base, se puede producir un error lógico conocido como ***recursividad infinita****, en donde se realizan llamadas recursivas en forma continua, hasta que se agota la memoria. Este error es análogo al problema de un ciclo infinito en una solución iterativa (sin recursividad).*

El método `mostrarFactoriales` (líneas 16 a 21) muestra los factoriales del 0 al 10. La llamada al método `factorial` ocurre en la línea 20. Este método recibe un parámetro de tipo `long` y devuelve un resultado de tipo `long`. La figura 15.4 prueba nuestros métodos `factorial` y `mostrarFactoriales`, llamando a `mostrarFactoriales` (línea 10). Los resultados de la figura 15.4 muestra que los valores de los factoriales crecen rápidamente. Utilizamos el tipo `long` (que puede representar enteros relativamente grandes) para que el programa pueda calcular factoriales mayores que 12!. Por desgracia, el método `factorial` produce valores grandes con tanta rapidez que los valores de los factoriales exceden pronto al valor máximo que puede almacenarse, incluso en una variable `long`.

Debido a las limitaciones de los tipos integrales, tal vez se necesiten variables `float` o `double` para calcular factoriales o números grandes. Esto apunta a una debilidad en la mayoría de los lenguajes de programación: a saber, que los lenguajes no se extienden fácilmente para manejar los requerimientos únicos de una aplicación. Como vimos en el capítulo 9, Java es un lenguaje extensible que nos permite crear números arbitrariamente grandes, si lo deseamos. De hecho, el paquete `java.math` cuenta con las clases `BigInteger` y `BigDecimal` explícitamente para los cálculos matemáticos de precisión arbitraria, que no pueden llevarse a cabo con los tipos primitivos. Para obtener más información acerca de estas clases, visite `java.sun.com/javase/6/docs/api/java/math/BigInteger.html` y `java.sun.com/javase/6/docs/api/java/math/BigDecimal.html`, respectivamente.

```
1  // Fig. 15.4: PruebaFactorial.java
2  // Prueba del método recursivo factorial.
3
4  public class PruebaFactorial
```

Figura 15.4 | Prueba del método `factorial`. (Parte 1 de 2).

```
{
   // calcula los factoriales del 0 al 10
   public static void main( String args[] )
   {
      CalculoFactorial calculoFactorial = new CalculoFactorial();
      calculoFactorial.mostrarFactoriales();
   } // fin del método main
} // fin de la clase PruebaFactorial
```

```
0! = 1
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
6! = 720
7! = 5040
8! = 40320
9! = 362880
10! = 3628800
```

Figura 15.4 | Prueba del método `factorial`. (Parte 2 de 2).

15.4 Ejemplo de uso de recursividad: serie de Fibonacci

La **serie de Fibonacci**,

0, 1, 1, 2, 3, 5, 8, 13, 21, …

empieza con 0 y 1, y tiene la propiedad de que cada número subsiguiente de Fibonacci es la suma de los dos números Fibonacci anteriores. Esta serie ocurre en la naturaleza y describe una forma de espiral. La proporción de números de Fibonacci sucesivos converge en un valor constante de 1.618…, un número denominado **proporción dorada**, o **media dorada**. Los humanos tienden a descubrir que la media dorada es estéticamente placentera. A menudo, los arquitectos diseñan ventanas, cuartos y edificios con una proporción de longitud-anchura en la que se utiliza la media dorada.

La serie de Fibonacci se puede definir de manera recursiva como:

fibonacci(0) = 0
fibonacci(1) = 1
fibonacci(n) = fibonacci(n – 1) + fibonacci(n – 2)

Observe que hay dos casos base para el cálculo de Fibonacci: `fibonacci(0)` se define como 0, y `fibonacci(1)` se define como 1. El programa de la figura 15.5 calcula el *i*-ésimo número de Fibonacci en forma recursiva, usando el método `fibonacci` (líneas 7 a 13). El método `mostrarFibonacci` (líneas 15 a 20) prueba a `fibonacci`, mostrando los valores de Fibonacci del 0 al 10. La variable `contador` creada en el encabezado de la instrucción `for` en la línea 17 indica cuál número de Fibonacci se debe calcular para cada iteración del número `for`. Los números de Fibonacci tienden a aumentar con rapidez. Por lo tanto, utilizamos `long` como el tipo del parámetro y el tipo de valor de retorno del método `fibonacci`. En la línea 9 de la figura 15.6 se hace una llamada al método `mostrarFibonacci` (línea 9) para calcular los valores de Fibonacci.

```
1  // Fig. 15.5: CalculoFibonacci.java
2  // Método fibonacci recursivo.
3
4  public class CalculoFibonacci
5  {
```

Figura 15.5 | Números de `Fibonacci` generados con un método recursivo. (Parte 1 de 2).

```
6      // declaración recursiva del método fibonacci
7      public long fibonacci( long numero )
8      {
9         if ( ( numero == 0 ) || ( numero == 1 ) ) // casos base
10           return numero;
11        else // paso recursivo
12           return fibonacci( numero - 1 ) + fibonacci( numero - 2 );
13     } // fin del método fibonacci
14
15     public void mostrarFibonacci()
16     {
17        for ( int contador = 0; contador <= 10; contador++ )
18           System.out.printf( "Fibonacci de %d es: %d\n", contador,
19              fibonacci( contador ) );
20     } // fin del método mostrarFibonacci
21  } // fin de la clase CalculoFibonacci
```

Figura 15.5 | Números de `Fibonacci` generados con un método recursivo. (Parte 2 de 2).

```
// Fig. 15.6: PruebaFibonacci.java
// Prueba del método recursivo fibonacci.

public class PruebaFibonacci
{
   public static void main( String args[] )
   {
      CalculoFibonacci calculoFibonacci = new CalculoFibonacci();
      calculoFibonacci.mostrarFibonacci();
   } // fin de main
} // fin de la clase PruebaFibonacci
```

```
Fibonacci de 0 es: 0
Fibonacci de 1 es: 1
Fibonacci de 2 es: 1
Fibonacci de 3 es: 2
Fibonacci de 4 es: 3
Fibonacci de 5 es: 5
Fibonacci de 6 es: 8
Fibonacci de 7 es: 13
Fibonacci de 8 es: 21
Fibonacci de 9 es: 34
Fibonacci de 10 es: 55
```

Figura 15.6 | Prueba del método `Fibonacci`.

La llamada al método `fibonacci` (línea 19 de la figura 15.5) desde `mostrarFibonacci` no es una llamada recursiva, pero todas las llamadas subsiguientes a `fibonacci` que se llevan a cabo desde el cuerpo de `fibonacci` (línea 12 de la figura 15.5) son recursivas, ya que en ese punto es el mismo método `fibonacci` el que inicia las llamadas. Cada vez que se hace una llamada a `fibonacci`, se evalúan inmediatamente los dos casos base: `numero` igual a 0 o `numero` igual a 1 (línea 9). Si esta condición es verdadera, `fibonacci` simplemente devuelve `numero` ya que `fibonacci(0)` es 0, y `fibonacci(1)` es 1. Lo interesante es que, si `numero` es mayor que 1, el paso recursivo genera *dos* llamadas recursivas (línea 12), cada una de ellas para un problema ligeramente más pequeño que el de la llamada original a `fibonacci`.

La figura 15.7 muestra cómo el método `fibonacci` evalúa `fibonacci( 3 )`. Observe que en la parte inferior de la figura, nos quedamos con los valores 1, 0 y 1; los resultados de evaluar los casos base. Los primeros dos valores de retorno (de izquierda a derecha), 1 y 0, se devuelven como los valores para las llamadas `fibonacci( 1 )` y `fibonacci( 0 )`. La suma 1 más 0 se devuelve como el valor de `fibonacci( 2 )`. Esto se suma al resultado (1)

de la llamada a fibonacci(1), para producir el valor 2. Después, este valor final se devuelve como el valor de fibonacci(3).

La figura 15.7 genera ciertas preguntas interesantes, en cuanto al orden en el que los compiladores de Java evalúan los operandos de los operadores. Este orden es distinto del orden en el que se aplican los operadores a sus operandos; a saber, el orden que dictan las reglas de la precedencia de operadores. De la figura 15.7, parece ser que mientras se evalúa fibonacci(3), se harán dos llamadas recursivas: fibonacci(2) y fibonacci(1). Pero ¿en qué orden se harán estas llamadas? El lenguaje Java especifica que el orden de evaluación de los operandos es de izquierda a derecha. Por ende, la llamada a fibonacci(2) se realiza primero, y después la llamada a fibonacci(1).

Hay que tener cuidado con los programas recursivos como el que utilizamos aquí para generar números de Fibonacci. Cada invocación del método fibonacci que no coincide con uno de los casos base (0 o 1) produce dos llamadas recursivas más al método fibonacci. Por lo tanto, este conjunto de llamadas recursivas se sale rápidamente de control. Para calcular el valor 20 de Fibonacci con el programa de la figura 15.5, se requieren 21,891 llamadas al método fibonacci; ¡para calcular el valor 30 de Fibonacci se requieren 2,692,537 llamadas! A medida que trate de calcular valores más grandes de Fibonacci, observará que cada número de Fibonacci consecutivo que calcule con la aplicación requiere un aumento considerable en tiempo de cálculo y en el número de llamadas al método fibonacci. Por ejemplo, el valor 31 de Fibonacci requiere 4,356,617 llamadas, y ¡el valor 32 de Fibonacci requiere 7,049,155 llamadas! Como puede ver, el número de llamadas al método fibonacci se incrementa con rapidez; 1,664,080 llamadas adicionales entre los valores 30 y 31 de Fibonacci, y ¡2,692,538 llamadas adicionales entre los valores 31 y 32 de Fibonacci! La diferencia en el número de llamadas realizadas entre los valores 31 y 32 de Fibonacci es de más de 1.5 veces la diferencia en el número de llamadas para los valores entre 30 y 31 de Fibonacci. Los problemas de esta naturaleza pueden humillar incluso hasta a las computadoras más poderosas del mundo. [*Nota:* en el campo de la teoría de la complejidad, los científicos de computadoras estudian qué tanto tienen que trabajar los algoritmos para completar sus tareas. Las cuestiones relacionadas con la complejidad se discuten con detalle en un curso del plan de estudios de ciencias computacionales de nivel superior, al que generalmente se le llama "Algoritmos". En el capítulo 16, Busqueda y ordenamiento, presentamos varias cuestiones acerca de la complejidad]. En los ejercicios le pediremos que mejore el programa de Fibonacci de la figura 15.5, de tal forma que calcule el monto aproximado de tiempo requerido para realizar el cálculo. Para este fin, llamará al método static de System llamado currentTimeMillis, el cual no recibe argumentos y devuelve el tiempo actual de la computadora en milisegundos.



Tip de rendimiento 15.1

Evite los programas recursivos al estilo de Fibonacci, ya que producen una "explosión" exponencial de llamadas a métodos.

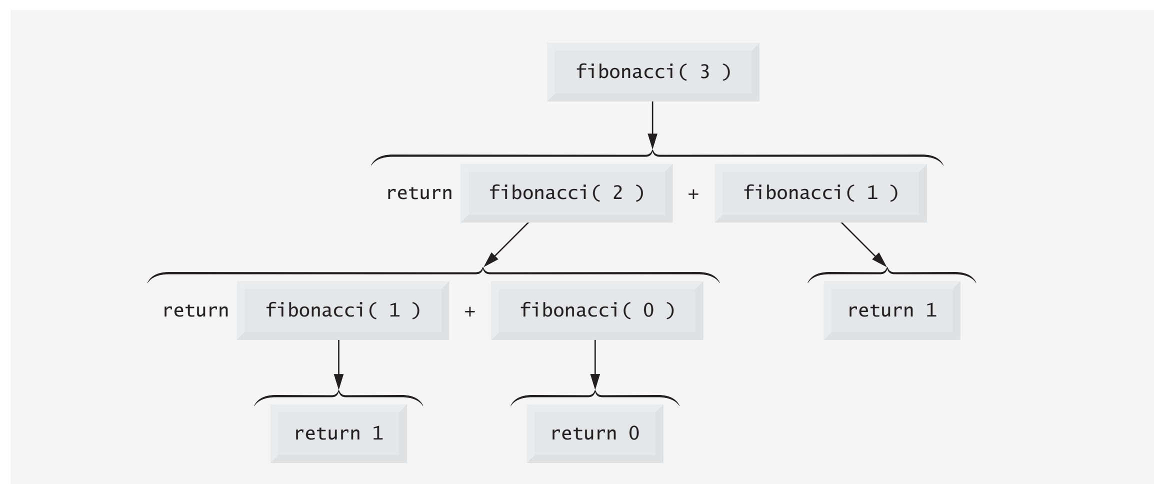


Figura 15.7 | Conjunto de llamadas recursivas para fibonacci (3).

15.5 La recursividad y la pila de llamadas a métodos

En el capítulo 6 se presentó la estructura de datos tipo pila, para comprender cómo Java realiza las llamadas a los métodos. Hablamos sobre la pila de llamadas a métodos (también conocida como la pila de ejecución del programa) y los registros de activación. En esta sección, utilizaremos estos conceptos para demostrar la forma en que la pila de ejecución del programa maneja las llamadas a los métodos recursivos.

Para empezar, regresemos al ejemplo de Fibonacci; en específico, la llamada al método `fibonacci` con el valor `3`, como en la figura 15.7. Para mostrar el orden en el que se colocan los registros de activación de las llamadas a los métodos en la pila, hemos clasificado las llamadas a los métodos con letras en la figura 15.8.

Cuando se hace la primera llamada al método (`A`), un registro de activación se mete en la pila de ejecución del programa, que contiene el valor de la variable local `numero` (3 en este caso). La pila de ejecución del programa, que incluye el registro de activación para la llamada `A` al método, se ilustra en la parte (a) de la figura 15.9. [*Nota:* aquí utilizamos una pila simplificada. En una computadora real, la pila de ejecución del programa y sus registros de activación serían más complejos que en la figura 15.9, ya que contienen información como la ubicación a la que va a regresar la llamada al método cuando haya terminado de ejecutarse].

Dentro de la llamada al método `A` se realizan las llamadas `B` y `E`. La llamada original al método no se ha completado, por lo que su registro de activación permanece en la pila. La primera llamada al método en realizarse desde el interior de `A` es la llamada `B` al método, por lo que se mete el registro de activación para la llamada `B` al método en la pila, encima del registro de activación para la llamada `A` al método. La llamada `B` al método debe ejecutarse y completarse antes de realizar la llamada `E`. Dentro de la llamada `B` al método, se harán las llamadas `C`

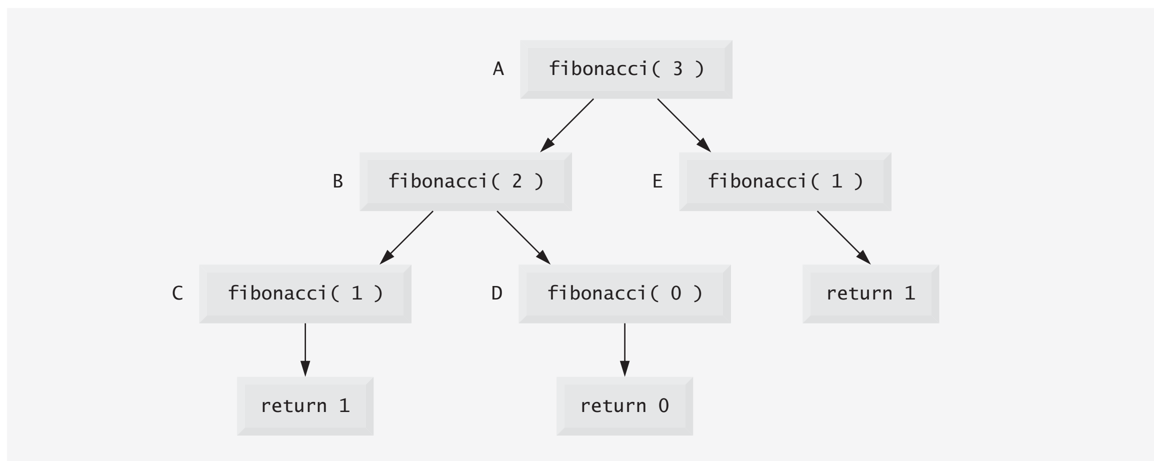


Figura 15.8 | Llamadas al método realizadas dentro de la llamada `fibonacci( 3 )`.

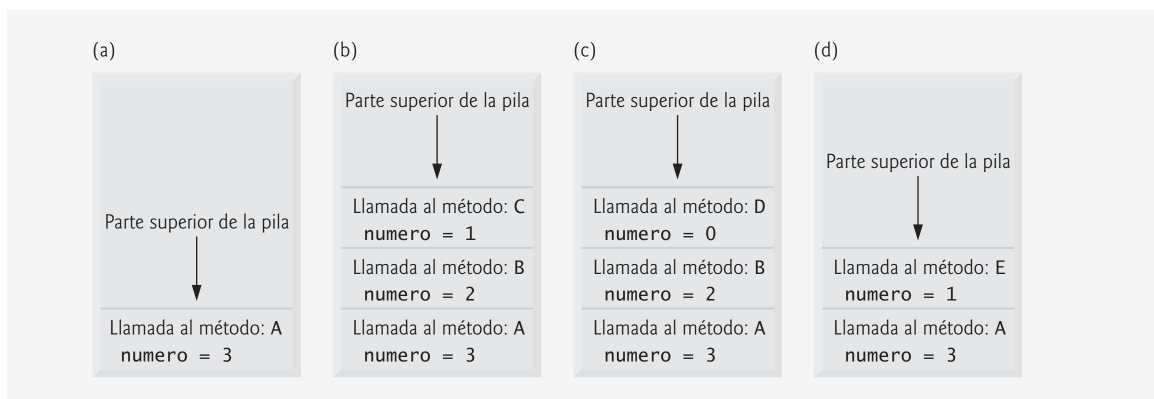


Figura 15.9 | Llamadas al método en la pila de ejecución del programa.

y D al método. La llamada C se realiza primero, y su registro de activación se mete en la pila [parte (b) de la figura 15.9]. La llamada B al método todavía no ha terminado, y su registro de activación sigue en la pila de llamadas a métodos. Cuando se ejecuta la llamada C, no realiza ninguna otra llamada al método, sino que simplemente devuelve el valor 1. Cuando este método regresa, su registro de activación se saca de la parte superior de la pila. La llamada al método en la parte superior de la pila es ahora B, que continúa ejecutándose y realiza la llamada D al método. El registro de activación para la llamada D se mete en la pila [parte (c) de la figura 15.9]. La llamada D al método se completa sin realizar ninguna otra llamada, y devuelve el valor 0. Después, se saca el registro de activación para esta llamada de la pila. Ahora, ambas llamadas al método que se realizaron desde el interior de la llamada B al método han regresado. La llamada B continúa ejecutándose, y devuelve el valor 1. La llamada B al método se completa y su registro de activación se saca de la pila. En este punto, el registro de activación para la llamada A al método se encuentra en la parte superior de la pila, y el método continúa su ejecución. Este método realiza la llamada E al método, cuyo registro de activación se mete ahora en la pila [parte (d) de la figura 15.9]. La llamada E al método se completa y devuelve el valor 1. El registro de activación para esta llamada al método se saca de la pila, y una vez más la llamada A al método continúa su ejecución. En este punto, la llamada A no realizará ninguna otra llamada al método y puede terminar su ejecución, para lo cual devuelve el valor 2 al método que llamó a A (fibonacci(3) = 2). El registro de activación de A se saca de la pila. Observe que el método en ejecución es siempre el que tiene su registro de activación en la parte superior de la pila, y el registro de activación para ese método contiene los valores de sus variables locales.

15.6 Comparación entre recursividad e iteración

En las secciones anteriores estudiamos los métodos `factorial` y `fibonacci`, que pueden implementarse fácilmente, ya sea en forma recursiva o iterativa. En esta sección compararemos los dos métodos, y veremos por qué le convendría al programador elegir un método en vez del otro, en una situación específica.

Tanto la iteración como la recursividad se basan en una instrucción de control: la iteración utiliza una instrucción de repetición (`for`, `while` o `do…while`), mientras que la recursividad utiliza una instrucción de selección (`if`, `if…else` o `switch`). Tanto la iteración como la recursividad implican la repetición: la iteración utiliza de manera explícita una instrucción de repetición, mientras que la recursividad logra la repetición a través de llamadas repetidas al método. La iteración y la recursividad implican una prueba de terminación: la iteración termina cuando falla la condición de continuación de ciclo, mientras que la recursividad termina cuando se llega a un caso base. La iteración con repetición controlada por contador y la recursividad llegan en forma gradual a la terminación: la iteración sigue modificando un contador, hasta que éste asume un valor que hace que falle la condición de continuación de ciclo, mientras que la recursividad sigue produciendo versiones cada vez más pequeñas del problema original, hasta que se llega a un caso base. Tanto la iteración como la recursividad pueden ocurrir infinitamente: un ciclo infinito ocurre con la iteración si la prueba de continuación de ciclo nunca se vuelve falsa, mientas que la recursividad infinita ocurre si el paso recursivo no reduce el problema cada vez, de forma tal que llegue a converger en el caso base, o si el caso base no se evalúa.

Para ilustrar las diferencias entre la iteración y la recursividad, examinaremos una solución iterativa para el problema del factorial (figuras 15.10 y 15.11). Observe que se utiliza una instrucción de repetición (líneas 12 y 13 de la figura 15.10), en vez de la instrucción de selección de la solución recursiva (líneas 9 a 12 de la figura 15.3). Observe que ambas soluciones usan una prueba de terminación. En la solución recursiva, en la línea 9 se evalúa el caso base. En la solución iterativa, en la línea 12 se evalúa la condición de continuación de ciclo; si la prueba falla, el ciclo termina. Por último, observe que en vez de producir versiones cada vez más pequeñas del problema original, la solución iterativa utiliza un contador que se modifica hasta que la condición de continuación de ciclo se vuelve falsa.

```
1  // Fig. 15.10: CalculoFactorial.java
2  // Método factorial iterativo.
3
4  public class CalculoFactorial
5  {
6     // declaración recursiva del método factorial
```

Figura 15.10 | Solución de factorial iterativa. (Parte 1 de 2).

```
    public long factorial( long numero )
    {
       long resultado = 1;

       // declaración iterativa del método factorial
       for ( long i = numero; i >= 1; i-- )
          resultado *= i;

       return resultado;
    } // fin del método factorial

    // muestra los factoriales para los valores del 0 al 10
    public void mostrarFactoriales()
    {
       // calcula los factoriales del 0 al 10
       for ( int contador = 0; contador <= 10; contador++ )
          System.out.printf( "%d! = %d\n", contador, factorial( contador ) );
    } // fin del método mostrarFactoriales
} // fin de la clase CalculoFactorial
```

Figura 15.10 | Solución de factorial iterativa. (Parte 2 de 2).

```
// Fig. 15.11: PruebaFactorial.java
// Prueba del método factorial iterativo.

public class PruebaFactorial
{
   // calcula los factoriales del 0 al 10
   public static void main( String args[] )
   {
      CalculoFactorial calculoFactorial = new CalculoFactorial();
      calculoFactorial.mostrarFactoriales();
   } // fin de main
} // fin de la clase PruebaFactorial
```

```
0! = 1
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
6! = 720
7! = 5040
8! = 40320
9! = 362880
10! = 3628800
```

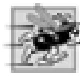
Figura 15.11 | Prueba de la solución de factorial iterativa.

La recursividad tiene muchas desventajas. Invoca al mecanismo en forma repetida, y en consecuencia se produce una sobrecarga de las llamadas al método. Esta repetición puede ser perjudicial, en términos de tiempo del procesador y espacio de la memoria. Cada llamada recursiva crea otra copia del método (en realidad, sólo las variables del mismo, que se almacenan en el registro de activación); este conjunto de copias puede consumir una cantidad considerable de espacio en memoria. Como la iteración ocurre dentro de un método, se evitan las llamadas repetidas al método y la asignación adicional de memoria. Entonces, ¿por qué elegir la recursividad?



Observación de ingeniería de software 15.1

Cualquier problema que se pueda resolver mediante la recursividad, se puede resolver también mediante la iteración (sin recursividad). Por lo general, se prefiere un método recursivo a uno iterativo cuando el primero refleja con más naturalidad el problema, y se produce un programa más fácil de entender y de depurar. A menudo, puede implementarse un método recursivo con menos líneas de código. Otra razón por la que es preferible elegir un método recursivo es que uno iterativo podría no ser aparente.



Tip de rendimiento 15.2

Evite usar la recursividad en situaciones en las que se requiera un alto rendimiento. Las llamadas recursivas requieren tiempo y consumen memoria adicional.



Error común de programación 15.2

Hacer que un método no recursivo se llame a sí mismo por accidente, ya sea en forma directa o indirecta a través de otro método, puede provocar recursividad infinita.

15.7 Las torres de Hanoi

En las secciones anteriores de este capítulo, estudiamos métodos que pueden implementarse con facilidad, tanto en forma recursiva como iterativa. En esta sección presentamos un problema cuya solución recursiva demuestra la elegancia de la recursividad, y cuya solución iterativa tal vez no sea tan aparente.

Las **torres de Hanoi** son uno de los problemas clásicos con los que todo científico computacional en ciernes tiene que lidiar. Cuenta la leyenda que en un templo del Lejano Oriente, los sacerdotes intentan mover una pila de discos dorados, de una aguja de diamante a otra (figura 15.12). La pila inicial tiene 64 discos insertados en una aguja y se ordenan de abajo hacia arriba, de mayor a menor tamaño. Los sacerdotes intentan mover la pila de una aguja a otra, con las restricciones de que sólo se puede mover un disco a la vez, y en ningún momento se puede colocar un disco más grande encima de uno más pequeño. Se cuenta con tres agujas, una de las cuales se utiliza para almacenar discos temporalmente. Se supone que el mundo acabará cuando los sacerdotes completen su tarea, por lo que hay pocos incentivos para que nosotros podamos facilitar sus esfuerzos.

Supongamos que los sacerdotes intentan mover los discos de la aguja 1 a la aguja 2. Deseamos desarrollar un algoritmo que imprima la secuencia precisa de transferencias de los discos de una aguja a otra.

Si tratamos de encontrar una solución iterativa, es probable que terminemos "atados" manejando los discos sin esperanza. En vez de ello, si atacamos este problema mediante la recursividad podemos producir rápidamente una solución. La acción de mover n discos puede verse en términos de mover sólo $n - 1$ discos (de ahí la recursividad) de la siguiente forma:

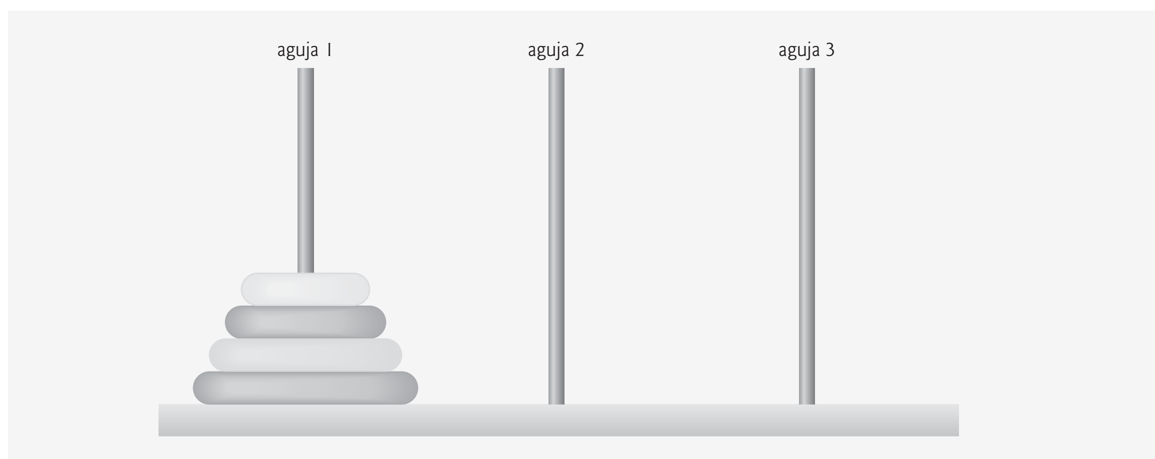


Figura 15.12 | Las torres de Hanoi para el caso con cuatro discos.

1. Mover n – 1 discos de la aguja 1 a la aguja 2, usando la aguja 3 como un área de almacenamiento temporal.
2. Mover el último disco (el más grande) de la aguja 1 a la aguja 3.
3. Mover n – 1 discos de la aguja 2 a la aguja 3, usando la aguja 1 como área de almacenamiento temporal.

El proceso termina cuando la última tarea implica mover n = 1 disco (es decir, el caso base). Esta tarea se logra con sólo mover el disco, sin necesidad de un área de almacenamiento temporal.

El programa de las figuras 15.13 y 15.14 resuelve las torres de Hanoi. En el constructor (líneas 9 a 12) se inicializa el número de discos a mover (`numDiscos`). El método `resolverTorres` (líneas 15 a 34) resuelve el acertijo de las torres de Hanoi, dado el número total de discos (en este caso 3), la aguja inicial, la aguja final y la aguja de almacenamiento temporal como parámetros. El caso base (líneas 19 a 23) ocurre cuando sólo se necesita mover un disco de la aguja inicial a la aguja final. En el paso recursivo (líneas 27 a 33), la línea 27 mueve `discos – 1` discos de la primera aguja (`agujaOrigen`) a la aguja de almacenamiento temporal (`agujaTemp`). Cuando se han movido todos los discos a la aguja temporal excepto uno, en la línea 30 se mueve el disco más grande de la aguja inicial a la aguja de destino. En la línea 33 se termina el resto de los movimientos, llamando al método `resolverTorres` para mover `discos – 1` discos de manera recursiva, de la aguja temporal (`agujaTemp`) a la aguja de destino (`agujaDestino`), esta vez usando la primera aguja (`agujaOrigen`) como aguja temporal.

```
1  // Fig. 15.13: TorresDeHanoi.java
2  // Programa que resuelve el problema de las torres de Hanoi, y
3  // demuestra la recursividad.
4
5  public class TorresDeHanoi
6  {
7     private int numDiscos; // número de discos a mover
8
9     public TorresDeHanoi( int discos )
10    {
11       numDiscos = discos;
12    } // fin del constructor de TorresDeHanoi
13
14    // mueve discos de una torre a otra, de manera recursiva
15    public void resolverTorres( int discos, int agujaOrigen, int agujaDestino,
16       int agujaTemp )
17    {
18       // caso base -- sólo hay que mover un disco
19       if ( discos == 1 )
20       {
21          System.out.printf( "\n%d --> %d", agujaOrigen, agujaDestino );
22          return;
23       } // fin de if
24
25       // paso recursivo -- mueve (disco - 1) discos de agujaOrigen
26       // a agujaTemp usando agujaDestino
27       resolverTorres( discos - 1, agujaOrigen, agujaTemp, agujaDestino );
28
29       // mueve el último disco de agujaOrigen a agujaDestino
30       System.out.printf( "\n%d --> %d", agujaOrigen, agujaDestino );
31
32       // mueve ( discos - 1 ) discos de agujaTemp a agujaDestino
33       resolverTorres( discos - 1, agujaTemp, agujaDestino, agujaOrigen );
34    } // fin del método resolverTorres
35 } // fin de la clase TorresDeHanoi
```

Figura 15.13 | Solución de las torres de Hanoi, con un método recursivo.

La figura 15.14 prueba nuestra solución de las torres de Hanoi. La línea 12 crea un objeto torres de Hanoi, pasando como parámetro el número total de los discos que se deben mover de una aguja a otra. La línea 15 llama al método recursivo `resolverTorres`, el cual muestra, al apuntador de comando, los pasos a seguir.

```
1  // Fig. 15.14: PruebaTorresDeHanoi.java
2  // Prueba la solución al problema de las torres de Hanoi.
3
4  public class PruebaTorresDeHanoi
5  {
6     public static void main( String args[] )
7     {
8        int agujaInicial = 1; // se usa el valor 1 para indicar agujaInicial en la salida
9        int agujaFinal = 3;   // se usa el valor 3 para indicar agujaFinal en la salida
10       int agujaTemp = 2;    // se usa el valor 2 para indicar agujaTemp en la salida
11       int totalDiscos = 3;  // número de discos
12       TorresDeHanoi torresDeHanoi = new TorresDeHanoi( totalDiscos );
13
14       // llamada no recursiva inicial: mueve todos los discos.
15       torresDeHanoi.resolverTorres( totalDiscos, agujaInicial, agujaFinal, agujaTemp );
16    } // fin de main
17 } // fin de la clase PruebaTorresDeHanoi
```

```
1 --> 3
1 --> 2
3 --> 2
1 --> 3
2 --> 1
2 --> 3
1 --> 3
```

Figura 15.14 | Prueba de la solución de las torres de Hanoi.

15.8 Fractales

Un **fractal** es una figura geométrica que se puede generar a partir de un patrón que se repite en forma recursiva (figura 15.15). Para modificar la figura, se aplica el patrón a cada segmento de la figura original. En esta sección analizaremos unas cuantas aproximaciones. [*Nota:* nos referiremos a nuestras figuras geométricas como fractales, aun cuando son aproximaciones]. Aunque estas figuras se han estudiado desde antes del siglo 20, fue el matemático polaco Benoit Mandelbrot quien introdujo el término "fractal" en la década de 1970, junto con los detalles específicos acerca de cómo se crea un fractal, y la aplicación práctica de los fractales. La geometría fractal de Mandelbrot proporciona modelos matemáticos para muchas formas complejas que se encuentran en la naturaleza, como las montañas, nubes y litorales. Los fractales tienen muchos usos en las matemáticas y la ciencia. Pueden utilizarse para comprender mejor los sistemas o patrones que aparecen en la naturaleza (por ejemplo, los ecosistemas), en el cuerpo humano (por ejemplo, en los pliegues del cerebro) o en el universo (por ejemplo, los grupos de galaxias). No todos los fractales se asemejan a los objetos en la naturaleza. El dibujo de fractales se ha convertido en una forma de arte popular. Los fractales tienen una **propiedad auto-similar**: cuando se subdividen en partes, cada una se asemeja a una copia del todo, en un tamaño reducido. Muchos fractales producen una copia exacta del original cuando se amplía una porción de la imagen original; se dice que dicho fractal es **estrictamente auto-similar**. En la sección 15.11 se proporcionan vínculos para diversos sitios Web en los que hay discusiones y demostraciones de los fractales.

Como ejemplo, veamos un fractal popular, estrictamente auto-similar, conocido como la **Curva de Koch** (figura 15.15). Para formar este fractal, se elimina la tercera parte media de cada línea en el dibujo, y se sustituye con dos líneas que forman un punto, de tal forma que si permaneciera la tercera parte media de la línea original, se formaría un triángulo equilátero. A menudo, las fórmulas para crear fractales implican eliminar toda, o parte de,

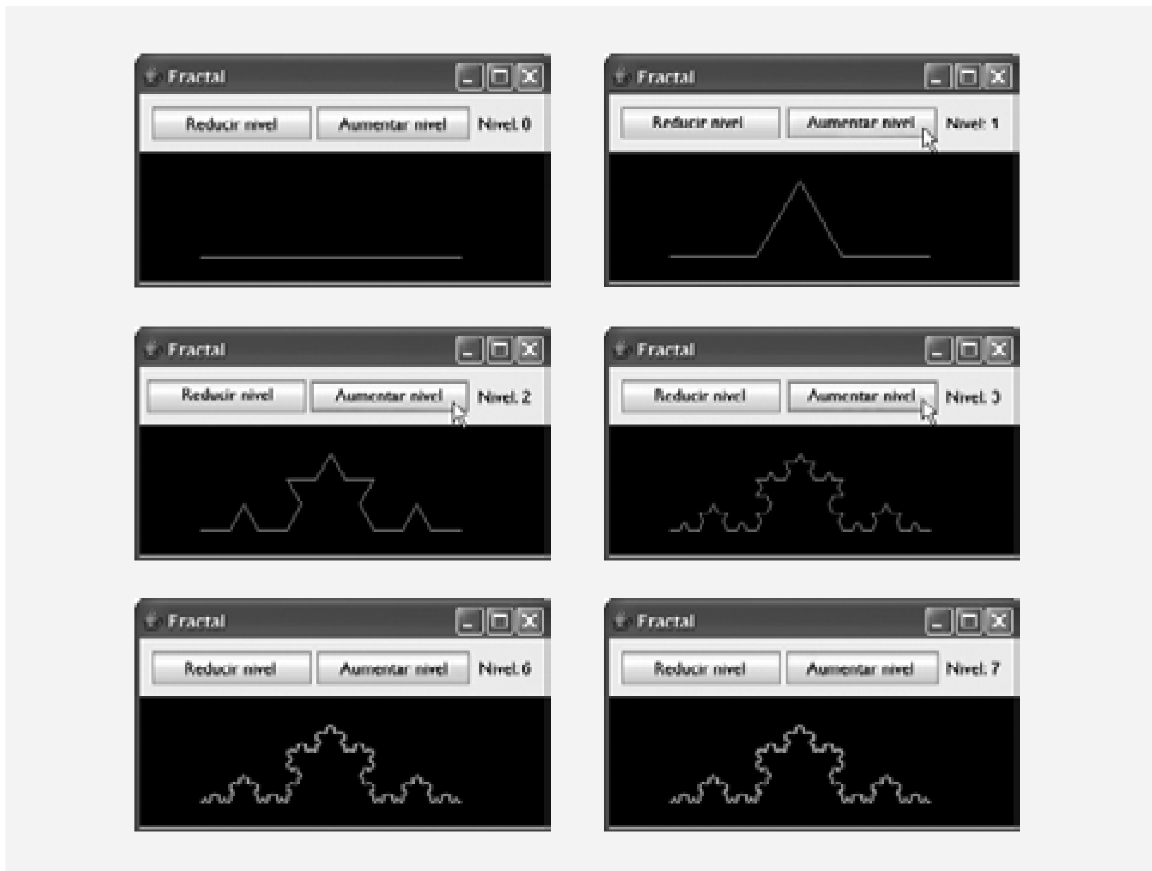


Figura 15.15 | Fractal `Curva de Koch`.

la imagen del fractal anterior. Este patrón ya se ha determinado para este fractal; en esta sección nos enfocaremos no sobre cómo determinar qué fórmulas se necesitan para un fractal específico, sino cómo utilizar esas fórmulas en una solución recursiva.

Empezamos con una línea recta [figura 15.15, parte (*a*)] y aplicamos el patrón, creando un triángulo a partir de la tercera parte media [figura 15.15, parte (*b*)]. Después aplicamos el patrón de nuevo a cada línea recta, lo cual produce la figura 15.15, parte (*c*). Cada vez que se aplica el patrón, decimos que el fractal está en un nuevo **nivel**, o **profundidad** (algunas veces se utiliza también el término **orden**). Los fractales pueden mostrarse en muchos niveles; por ejemplo, a un fractal de nivel 3 se le han aplicado tres iteraciones del patrón [figura 15.15, partes (*e* y *f*)]. Como éste es un fractal estrictamente auto-similar, cada porción del mismo contiene una copia exacta del fractal. Por ejemplo, en la parte (*f*) de la figura 15.15, hemos resaltado una porción del fractal con un cuadro color rojo punteado. Si se aumentara el tamaño de la imagen en este cuadro, se vería exactamente igual que el fractal completo de la parte (*f*).

Hay un fractal similar, llamado **Copo de nieve de Koch**, que es similar a la Curva de Koch, pero empieza con un triángulo en vez de una línea. Se aplica el mismo patrón a cada lado del triángulo, lo cual produce una imagen que se asemeja a un copo de nieve encerrado. Hemos optado por enfocarnos en la Curva de Koch por cuestión de simpleza. Para aprender más acerca de la Curva de Koch y del Copo de nieve de Koch, vea los vínculos de la sección 15.11.

Ahora demostraremos el uso de la recursividad para dibujar fractales, escribiendo un programa para crear un fractal estrictamente auto-similar. A este fractal lo llamaremos “fractal Lo”, en honor de Sin Han Lo, un colega de Deitel & Associates que lo creó. En un momento dado, el fractal se asemejará a la mitad de una pluma (vea los resultados en la figura 15.22). El caso base, o nivel 0 del fractal, empieza como una línea entre dos puntos, `A` y `B` (figura 15.16). Para crear el siguiente nivel superior, buscamos el punto medio (`C`) de la línea. Para calcular

la ubicación del punto C, utilice la siguiente fórmula: [*Nota:* la x y la y a la izquierda de cada letra se refieren a las coordenadas *x* y *y* de ese punto, respectivamente. Por ejemplo, xA se refiere a la coordenada x del punto A, mientras que yC se refiere a la coordenada y del punto C. En nuestros diagramas denotamos el punto por su letra, seguida de dos números que representan las coordenadas *x* y *y*].

```
xC = (xA + xB) / 2;
yC = (yA + yB) / 2;
```

Para crear este fractal, también debemos buscar un punto D que se encuentre a la izquierda del segmento AC y que cree un triángulo recto isósceles ADC. Para calcular la ubicación del punto D, utilice las siguientes fórmulas:

```
xD = xA + (xC - xA) / 2 - (yC - yA) / 2;
yD = yA + (yC - yA) / 2 + (xC - xA) / 2;
```

Ahora nos movemos del nivel 0 al nivel 1 de la siguiente manera: primero, se suman los puntos C y D (como en la figura 15.17). Después se elimina la línea original y se agregan los segmentos DA, DC y DB. El resto de las líneas se curvearán en un ángulo, haciendo que nuestro fractal se vea como una pluma. Para el siguiente nivel del fractal, este algoritmo se repite en cada una de las tres líneas en el nivel 1. Para cada línea, se aplican las fórmulas anteriores, en donde el punto anterior D se considera ahora como el punto A, mientras que el otro extremo de cada línea se considera como el punto B. La figura 15.18 contiene la línea del nivel 0 (ahora una línea punteada) y las tres líneas que se agregaron del nivel 1. Hemos cambiado el punto D para que sea el punto A, y los puntos originales A, C y B son B1, B2 y B3, respectivamente. Las fórmulas anteriores se han utilizado para buscar los nuevos puntos C y D en cada línea. Estos puntos también se enumeran del 1 al 3 para llevar la cuenta de cuál punto está asociado con cada línea. Por ejemplo, los puntos C1 y D1 representan a los puntos C y D asociados con la línea que se forma de los puntos A a B1. Para llegar al nivel 2, se eliminan las tres líneas de la figura 15.18 y se sustituyen con nuevas líneas de los puntos C y D que se acaban de agregar. La figura 15.19 muestra las nuevas líneas (las líneas del nivel 2 se muestran como líneas punteadas, para conveniencia del lector). La figura 15.20 muestra el nivel 2 sin las líneas punteadas del nivel 1. Una vez que se ha repetido este proceso varias veces, el fractal creado empezará a parecerse a la mitad de una pluma, como se muestra en los resultados de la figura 15.22. En breve presentaremos el código para esta aplicación.

La aplicación de la figura 15.21 define la interfaz de usuario para dibujar este fractal (que se muestra al final de la figura 15.22). La interfaz consiste de tres botones: uno para que el usuario modifique el color del fractal, otro para incrementar el nivel de recursividad y uno para reducir el nivel de recursividad. Un objeto JLabel lleva la cuenta del nivel actual de recursividad, que se modifica mediante una llamada al método establecerNivel, que veremos en breve. En las líneas 15 y 16 se especifican las constantes ANCHURA y ALTURA como 400 y 480 respectivamente, para indicar el tamaño del objeto JFrame. El color predeterminado para dibujar el fractal será azul (línea 18). El usuario activa un evento ActionEvent haciendo clic en el botón Color. El manejador de eventos para este botón se registra en las líneas 38 a 54. El método actionPerformed muestra un cuadro de diálogo JColorChoo-

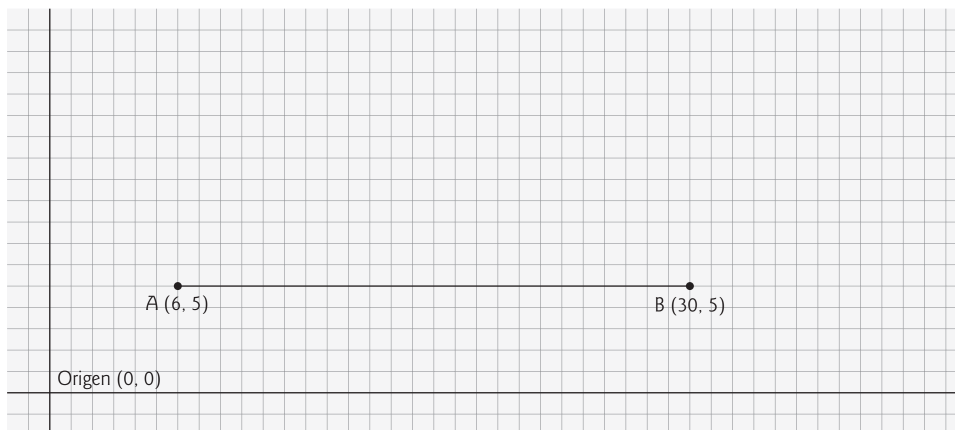


Figura 15.16 | El "fractal Lo" en el nivel 0.

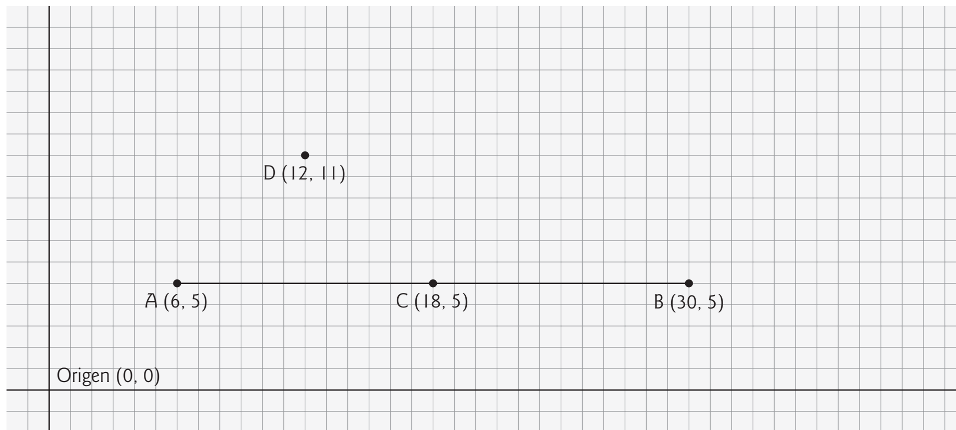


Figura 15.17 | Determinación de los puntos C y D para el nivel 1 del "fractal Lo".

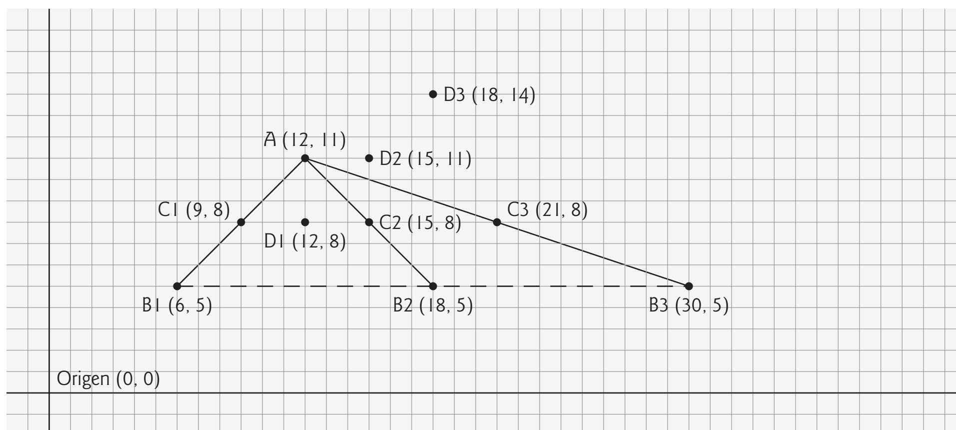


Figura 15.18 | El "fractal Lo" en el nivel 1, y se determinan los puntos C y D para el nivel 2. [*Nota:* se incluye el fractal en el nivel 0 como una línea punteada, para recordar en dónde se encontraba la línea en relación con el fractal actual].

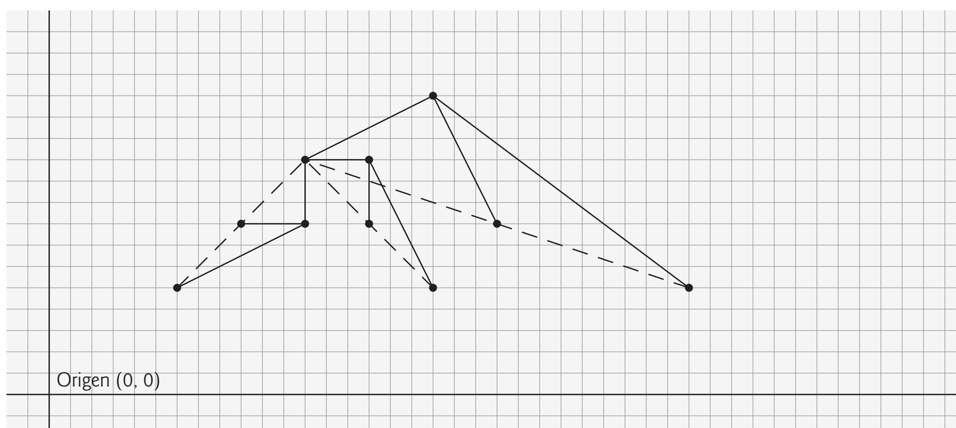


Figura 15.19 | El "fractal Lo" en el nivel 2, y se proporcionan las líneas punteadas del nivel 1.

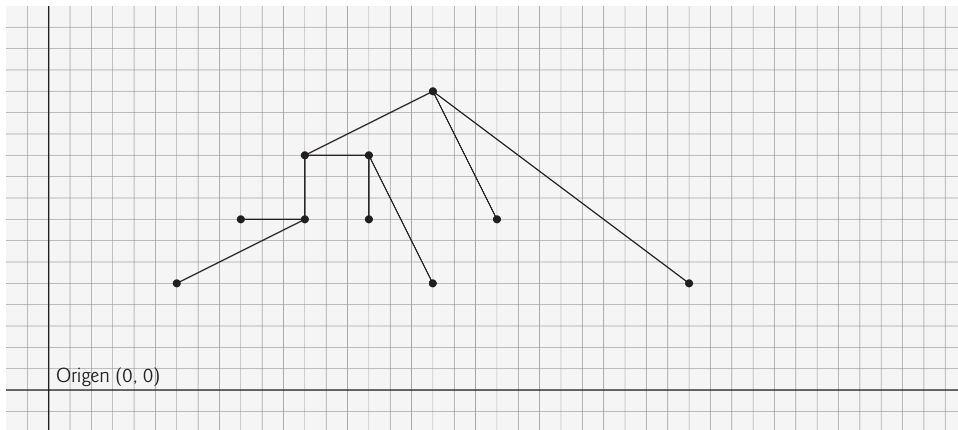


Figura 15.20 | El "fractal Lo" en el nivel 2.

`ser`. Este cuadro de diálogo devuelve el objeto **Color** seleccionado, o azul (si el usuario oprime **Cancelar** o cierra el cuadro de diálogo sin oprimir **Aceptar**). En la línea 51 se hace una llamada al método `establecerColor` en la clase `FractalJPanel` para actualizar el color.

El manejador de eventos para el botón **Reducir nivel** se registra en las líneas 60 a 78. En el método `actionPerformed`, en las líneas 66 y 67 obtienen el nivel actual de recursividad y lo reducen en 1. En la línea 70 se

```
1  // Fig. 15.21: Fractal.java
2  // Demuestra la interfaz de usuario para dibujar un fractal.
3  import java.awt.Color;
4  import java.awt.FlowLayout;
5  import java.awt.event.ActionEvent;
6  import java.awt.event.ActionListener;
7  import javax.swing.JFrame;
8  import javax.swing.JButton;
9  import javax.swing.JLabel;
10 import javax.swing.JPanel;
11 import javax.swing.JColorChooser;
12
13 public class Fractal extends JFrame
14 {
15    private final int ANCHURA = 400;     // define la anchura de la GUI
16    private final int ALTURA = 480;      // define la altura de la GUI
17    private final int NIVEL_MIN = 0, NIVEL_MAX = 15;
18    private Color color = Color.BLUE;
19
20    private JButton cambiarColorJButton, aumentarNivelJButton,
21       reducirNivelJButton;
22    private JLabel nivelJLabel;
23    private FractalJPanel espacioDibujo;
24    private JPanel principalJPanel, controlJPanel;
25
26    // establece la GUI
27    public Fractal()
28    {
29       super( "Fractal" );
30
```

Figura 15.21 | Demostración de la interfaz de usuario del fractal. (Parte 1 de 3).

```
        // establece el panel de control
        controlJPanel = new JPanel();
        controlJPanel.setLayout( new FlowLayout() );

        // establece el botón de color y registra el componente de escucha
        cambiarColorJButton = new JButton( "Color" );
        controlJPanel.add( cambiarColorJButton );
        cambiarColorJButton.addActionListener(
           new ActionListener() // clase interna anónima
           {
              // procesa el evento de cambiarColorJButton
              public void actionPerformed( ActionEvent evento )
              {
                 color = JColorChooser.showDialog(
                    Fractal.this, "Elija un color", color );

                 // establece el color predeterminado, si no se devuelve un color
                 if ( color == null )
                    color = Color.BLUE;

                 espacioDibujo.establecerColor( color );
              } // fin del método actionPerformed
           } // fin de la clase interna anónima
        ); // fin de addActionListener

        // establece botón para reducir nivel, para agregarlo al panel de control y
        // registra el componente de escucha
        reducirNivelJButton = new JButton( "Reducir nivel" );
        controlJPanel.add( reducirNivelJButton );
        reducirNivelJButton.addActionListener(
           new ActionListener() // clase interna anónima
           {
              // procesa el evento de reducirNivelJButton
              public void actionPerformed( ActionEvent evento )
              {
                 int nivel = espacioDibujo.obtenerNivel();
                 nivel--; // reduce el nivel en uno

                 // modifica el nivel si es posible
                 if ( ( nivel >= NIVEL_MIN ) && ( nivel <= NIVEL_MAX ) )
                 {
                    nivelJLabel.setText( "Nivel: " + nivel );
                    espacioDibujo.establecerNivel( nivel );
                    repaint();
                 } // fin de if
              } // fin del método actionPerformed
           } // fin de la clase interna anónima
        ); // fin de addActionListener

        // establece el botón para aumentar nivel, para agregarlo al panel de control
        // y registra el componente de escucha
        aumentarNivelJButton = new JButton( "Aumentar nivel" );
        controlJPanel.add( aumentarNivelJButton );
        aumentarNivelJButton.addActionListener(
           new ActionListener() // clase interna anónima
           {
              // procesa el evento de aumentarNivelJButton
              public void actionPerformed( ActionEvent evento )
```

Figura 15.21 | Demostración de la interfaz de usuario del fractal. (Parte 2 de 3).

```
89             {
90                int nivel = espacioDibujo.obtenerNivel();
91                nivel++; // aumenta el nivel en uno
92
93                // modifica el nivel si es posible
94                if ( ( nivel >= NIVEL_MIN ) && ( nivel <= NIVEL_MAX ) )
95                {
96                   nivelJLabel.setText( "Nivel: " + nivel );
97                   espacioDibujo.establecerNivel( nivel );
98                   repaint();
99                } // fin de if
100            } // fin del método actionPerformed
101         } // fin de la clase interna anónima
102      ); // fin de addActionListener
103
104      // establece nivelJLabel para agregarlo a controlJPanel
105      nivelJLabel = new JLabel( "Nivel: 0" );
106      controlJPanel.add( nivelJLabel );
107
108      espacioDibujo = new FractalJPanel( 0 );
109
110      // crea principalJPanel para que contenga a controlJPanel y espacioDibujo
111      principalJPanel = new JPanel();
112      principalJPanel.add( controlJPanel );
113      principalJPanel.add( espacioDibujo );
114
115      add( principalJPanel ); // agrega JPanel a JFrame
116
117      setSize( ANCHURA, ALTURA ); // establece el tamaño de JFrame
118      setVisible( true ); // muestra JFrame
119   } // fin del constructor de Fractal
120
121   public static void main( String args[] )
122   {
123      Fractal demo = new Fractal();
124      demo.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
125   } // fin de main
126 } // fin de la clase Fractal
```

Figura 15.21 | Demostración de la interfaz de usuario del fractal. (Parte 3 de 3).

realiza una verificación, para asegurar que el nivel sea mayor o igual que 0 (`NIVEL_MIN`); el fractal no está definido para cualquier nivel de recursividad menor que 0. El programa permite al usuario avanzar hacia cualquier nivel deseado, pero en cierto punto (nivel 10 y superior en este ejemplo) el despliegue del fractal se vuelve cada vez más lento, ya que hay muchos detalles que dibujar. En las líneas 72 a 74 se restablece la etiqueta del nivel para reflejar el cambio; se establece el nuevo nivel y se hace una llamada al método `repaint` para actualizar la imagen y mostrar el fractal correspondiente al nuevo nivel.

El objeto `JButton` **Aumentar nivel** funciona de la misma forma que el objeto `JButton` **Reducir nivel**, excepto que el nivel se incrementa en vez de reducirse para mostrar más detalles del fractal (líneas 90 y 91). Cuando se ejecuta la aplicación por primera vez, el nivel se establece en 0, en donde se muestra una línea azul entre dos puntos especificados en la clase `FractalJPanel`.

La clase `FractalJPanel` de la figura 15.22 especifica las medidas del objeto `JPanel` del dibujo como `400` por `400` (líneas 13 y 14). El constructor de `FractalJPanel` (líneas 18 a 24) recibe el nivel actual como parámetro y lo asigna a su variable de instancia `nivel`. La variable de instancia `color` se establece en el color azul predeterminado. En las líneas 22 y 23 se cambia el color de fondo del objeto `JPanel` para que sea blanco (para la visibilidad de los colores utilizados para dibujar el fractal), y se establecen las nuevas medidas del objeto `JPanel`, en donde se dibujará el fractal.

```
// Fig. 15.22: FractalJPanel.java
// FractalJPanel demuestra el dibujo recursivo de un fractal.
import java.awt.Graphics;
import java.awt.Color;
import java.awt.Dimension;
import javax.swing.JPanel;

public class FractalJPanel extends JPanel
{
   private Color color; // almacena el color utilizado para dibujar el fractal
   private int nivel;   // almacena el nivel actual del fractal

   private final int ANCHURA = 400;  // define la anchura de JPanel
   private final int ALTURA = 400;   // define la altura de JPanel

   // establece el nivel inicial del fractal al valor especificado
   // y establece las especificaciones del JPanel
   public FractalJPanel( int nivelActual )
   {
      color = Color.BLUE;  // inicializa el color de dibujo en azul
      nivel = nivelActual; // establece el nivel inicial del fractal
      setBackground( Color.WHITE );
      setPreferredSize( new Dimension( ANCHURA, ALTURA ) );
   } // fin del constructor de FractalJPanel

   // dibuja el fractal en forma recursiva
   public void dibujarFractal( int nivel, int xA, int yA, int xB,
      int yB, Graphics g )
   {
      // caso base: dibuja una línea que conecta dos puntos dados
      if ( nivel == 0 )
         g.drawLine( xA, yA, xB, yB );
      else // paso recursivo: determina los nuevos puntos, dibuja el siguiente nivel
      {
         // calcula punto medio entre (xA, yA) y (xB, yB)
         int xC = ( xA + xB ) / 2;
         int yC = ( yA + yB ) / 2;

         // calcula el cuarto punto (xD, yD) que forma un
         // triángulo recto isósceles entre (xA, yA) y (xC, yC)
         // en donde el ángulo recto está en (xD, yD)
         int xD = xA + ( xC - xA ) / 2 - ( yC - yA ) / 2;
         int yD = yA + ( yC - yA ) / 2 + ( xC - xA ) / 2;

         // dibuja el Fractal en forma recursiva
         dibujarFractal( nivel - 1, xD, yD, xA, yA, g );
         dibujarFractal( nivel - 1, xD, yD, xC, yC, g );
         dibujarFractal( nivel - 1, xD, yD, xB, yB, g );
      } // fin de else
   } // fin del método dibujarFractal

   // inicia el dibujo del fractal
   public void paintComponent( Graphics g )
   {
      super.paintComponent( g );

      // dibuja el patrón del fractal
      g.setColor( color );
      dibujarFractal( nivel, 100, 90, 290, 200, g );
```

Figura 15.22 | Dibujo del "fractal Lo" mediante el uso de la recursividad. (Parte 1 de 3).

```
   } // fin del método paintComponent

   // establece el color de dibujo a c
   public void establecerColor( Color c )
   {
      color = c;
   } // fin del método setColor

   // establece el nuevo nivel de recursividad
   public void establecerNivel( int nivelActual )
   {
      nivel = nivelActual;
   } // fin del método setLevel

   // devuelve el nivel de recursividad
   public int obtenerNivel()
   {
      return nivel;
   } // fin del método getLevel
} // fin de la clase FractalJPanel
```

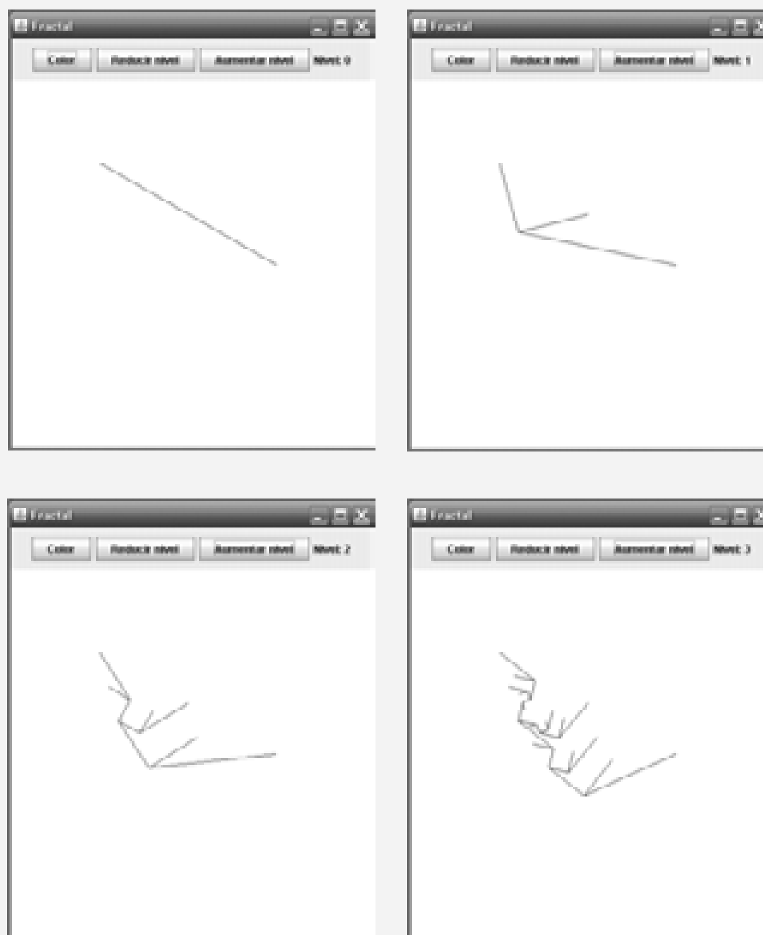


Figura 15.22 | Dibujo del "fractal Lo" mediante el uso de la recursividad. (Parte 2 de 3).

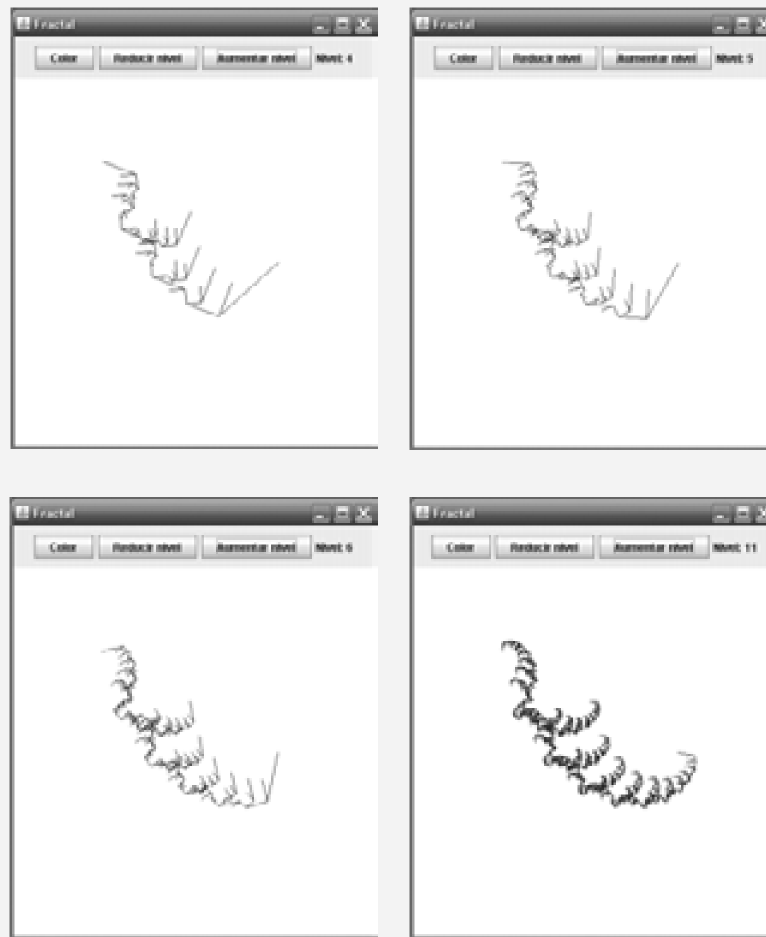


Figura 15.22 | Dibujo del “fractal Lo” mediante el uso de la recursividad. (Parte 3 de 3).

En las líneas 27 a 50 se define el método recursivo que crea el fractal. Este método recibe seis parámetros: el nivel, cuatro enteros que especifican las coordenadas *x* y *y* de dos puntos, y el objeto `g` de `Graphics`. El caso base para este método (línea 31) ocurre cuando `nivel` es igual a 0, en cuyo momento se dibujará una línea entre los dos puntos que se proporcionan como parámetros. En las líneas 36 a 43 se calcula (`xC`, `yC`), el punto medio entre (`xA`, `yA`) y (`xB`, `yB`), y (`xD`, `yD`), el punto que crea un triángulo isósceles recto con (`xA`, `yA`) y (`xC`, `yC`). En las líneas 46 a 48 se realizan tres llamadas recursivas en tres conjuntos distintos de puntos.

En el método `paintComponent`, en la línea 59 se realiza la primera llamada al método `dibujarFractal` para empezar el dibujo. Esta llamada al método no es recursiva, pero todas las llamadas subsiguientes a `dibujarFractal` que se realicen desde el cuerpo de `dibujarFractal` sí lo son. Como las líneas no se dibujarán sino hasta que se llegue al caso base, la distancia entre dos puntos se reduce en cada llamada recursiva. A medida que aumenta el nivel de recursividad, el fractal se vuelve más uniforme y detallado. La figura de este fractal se estabiliza a medida que el nivel se acerca a 11. Los fractales se estabilizarán en distintos niveles, con base en la figura y el tamaño del fractal.

En la figura 15.22 se muestra el desarrollo del fractal, de los niveles 0 al 6. La última imagen muestra la figura que define el fractal en el nivel 11. Si nos enfocamos en uno de los brazos de este fractal, será idéntico a la imagen completa. Esta propiedad define al fractal como estrictamente auto-similar. En la sección 15.11 podrá consultar más recursos acerca de los fractales.

15.9 "Vuelta atrás" recursiva (backtracking)

Todos nuestros métodos recursivos tienen una arquitectura similar; si se llega al caso base, devuelven un resultado; si no, hacen una o más llamadas recursivas. En esta sección exploraremos un método recursivo más completo, que busca una ruta a través de un laberinto, y devuelve verdadero si hay una posible solución al laberinto. La solución implica recorrer el laberinto un paso a la vez, en donde los movimientos pueden ser hacia abajo, a la derecha, hacia arriba o a la izquierda (no se permiten movimientos diagonales). De la posición actual en el laberinto (empezando con el punto de entrada), se realizan los siguientes pasos: se elije una dirección, se realiza el movimiento en esa dirección y se hace una llamada recursiva para resolver el resto del laberinto desde la nueva ubicación. Cuando se llega a un punto sin salida (es decir, no podemos avanzar más pasos sin pegar en la pared), retrocedemos a la ubicación anterior y tratamos de avanzar en otra dirección. Si no puede elegirse otra dirección, retrocedemos de nuevo. Este proceso continúa hasta que encontramos un punto en el laberinto en donde puede realizarse un movimiento en otra dirección. Una vez que se encuentra dicha ubicación, avanzamos en la nueva dirección y continuamos con otra llamada recursiva para resolver el resto del laberinto.

Para retroceder a la ubicación anterior en el laberinto, nuestro método recursivo simplemente devuelve falso, avanzando hacia arriba en la cadena de llamadas a métodos, hasta la llamada recursiva anterior (que hace referencia a la ubicación anterior en el laberinto). A este proceso de utilizar la recursividad para regresar a un punto de decisión anterior se le conoce como **"vuelta atrás" recursiva**. Si un conjunto de llamadas recursivas no resulta en una solución para el problema, el programa retrocede hasta el punto de decisión anterior y toma una decisión distinta, lo que a menudo produce otro conjunto de llamadas recursivas. En este ejemplo, el punto de decisión anterior es la ubicación anterior en el laberinto, y la decisión a realizar es la dirección que debe tomar el siguiente movimiento. Una dirección ha conducido a un punto sin salida, por lo que la búsqueda continúa con una dirección diferente. A diferencia de nuestros demás programas recursivos, que llegaron al caso base y luego regresaron a través de toda la cadena de llamadas a métodos, hasta la llamada al método original, la solución de "vuelta atrás" recursiva para el problema del laberinto utiliza la recursividad para regresar sólo una parte a través de la cadena de llamadas a métodos, y después probar una dirección diferente. Si la vuelta atrás llega a la ubicación de entrada del laberinto y se han recorrido todas las direcciones, entonces el laberinto no tiene solución.

En los ejercicios del capítulo le pediremos que implemente soluciones de "vuelta atrás" recursivas para el problema del laberinto (ejercicios 15.20, 15.21 y 15.22) y para el problema de las Ocho Reinas (ejercicio 15.15), el cual trata de encontrar la manera de colocar ocho reinas en un tablero de ajedrez vacío, de forma que ninguna reina esté "atacando" a otra (es decir, que no haya dos reinas en la misma fila, en la misma columna o a lo largo de la misma diagonal). En la sección 15.11 podrá consultar vínculos hacia más información sobre la "vuelta atrás" recursiva.

15.10 Conclusión

En este capítulo aprendió a crear métodos recursivos; es decir, métodos que se llaman a sí mismos. Aprendió que los métodos recursivos generalmente dividen a un problema en dos piezas conceptuales: una pieza que el método sabe cómo resolver (el caso base) y una pieza que el método no sabe cómo resolver (el paso recursivo). El paso recursivo es una versión ligeramente más pequeña del problema original, y se lleva a cabo mediante una llamada a un método recursivo. Vio algunos ejemplos populares de recursividad, incluyendo el cálculo de factoriales y la producción de valores en la serie de Fibonacci. Después aprendió cómo funciona la recursividad "detrás de las cámaras", incluyendo el orden en el que se meten o se sacan las llamadas a métodos recursivos de la pila de ejecución del programa. Después comparó los métodos recursivo e iterativo (no recursivo). Aprendió a resolver un problema más complejo mediante la recursividad: mostrar fractales. El capítulo concluyó con una introducción a la "vuelta atrás" recursiva, una técnica para resolver problemas que implica retroceder a través de llamadas recursivas para probar distintas soluciones posibles. En el siguiente capítulo, aprenderá diversas técnicas para ordenar listas de datos y buscar un elemento en una lista de datos, y bajo qué circunstancias debe utilizarse cada técnica de búsqueda y ordenamiento.

15.11 Recursos en Internet y Web

Conceptos de recursividad

`en.wikipedia.org/wiki/Recursion`

Artículo de Wikipedia, que proporciona los fundamentos de la recursividad y varios recursos para los estudiantes.

es.wikipedia.org/wiki/Recursividad
El mismo artículo anterior de Wikipedia, en español.

www.cafeaulait.org/javatutorial.html
Proporciona una breve introducción a la recursividad en Java, y también cubre otros temas relacionados con Java.

Pilas

www.cs.auc.dk/~normark/eciu-recursion/html/recit-slide-implerec.html
Proporciona diapositivas acerca de la implementación de la recursividad mediante el uso de pilas.

faculty.juniata.edu/kruse/cs2java/recurimpl.htm
Proporciona un diagrama de la pila de ejecución del programa y describe la forma en que funciona la pila.

Fractales

math.rice.edu/~lanius/frac/
Proporciona ejemplos de otros fractales, como el Copo de nieve de Koch, el Triángulo de Sierpinski y los fractales de Parque Jurásico.

www.lifesmith.com/
Proporciona cientos de imágenes de fractales coloridas, junto con una explicación detallada acerca de los conjuntos de Mandelbrot y Julia, dos conjuntos comunes de fractales.

www.jracademy.com/~jtucek/math/fractals.html
Contiene dos películas AVI creadas al realizar acercamientos continuos en los fractales, conocidos como los conjuntos de ecuaciones de Mandelbrot y Julia.

www.faqs.org/faqs/fractal-faq/
Proporciona las respuestas a muchas preguntas acerca de los fractales.

spanky.triumf.ca/www/fractint/fractint.html
Contiene vínculos para descargar Fractint, un programa de freeware para generar fractales.

www.42explore.com/fractal.htn
Proporciona una lista de URLs en fractales y herramientas de software que crean fractales.

www.arcytech.org/java/fractals/koch.shtml
Proporciona una introducción detallada al fractal de la Curva de Koch y un applet que demuestra el fractal.

library.thinkquest.org/26688/koch.html
Muestra un applet de la Curva de Koch, y proporciona el código fuente.

"Vuelta atrás" recursiva

www.cs.sfu.ca/CourseCentral/201/havens/notes/Lecture14.pdf
Proporciona una breve introducción a la "vuelta atrás" recursiva, incluyendo un ejemplo acerca de la planeación de una ruta de viaje.

math.hws.edu/xJava/PentominosSolver
Proporciona un programa que utiliza la "vuelta atrás" recursiva para resolver un problema, conocido como el acertijo de Pentominós (que se describe en el sitio).

Resumen

Sección 15.1 Introducción

- Un método recursivo se llama a sí mismo en forma directa o indirecta a través de otro método.
- Cuando se llama a un método recursivo para resolver un problema, en realidad el método es capaz de resolver sólo el (los) caso(s) más simple(s), o caso(s) base. Si se llama con un caso base, el método devuelve un resultado.

Sección 15.2 Conceptos de recursividad

- Si se llama a un método recursivo con un problema más complejo que el caso base, por lo general, divide el problema en dos piezas conceptuales: una pieza que el método sabe cómo resolver y otra pieza que no sabe cómo resolver.

- Para que la recursividad sea factible, la pieza que el método no sabe cómo resolver debe asemejarse al problema original, pero debe ser una versión ligeramente más simple o pequeña del mismo. Como este nuevo problema se parece al problema original, el método llama a una nueva copia de sí mismo para trabajar en el problema más pequeño; a esto se le conoce como paso recursivo.
- Para que la recursividad termine en un momento dado, cada vez que un método se llame a sí mismo con una versión más simple del problema original, la secuencia de problemas cada vez más pequeños debe converger en un caso base. Cuando el método reconoce el caso base, devuelve un resultado a la copia anterior del método.
- Una llamada recursiva puede ser una llamada a otro método, que a su vez realiza una llamada de vuelta al método original. Dicho proceso sigue provocando una llamada recursiva al método original. A esto se le conoce como llamada recursiva indirecta, o recursividad indirecta.

Sección 15.3 Ejemplo de uso de recursividad: factoriales

- La acción de omitir el caso base, o escribir el paso recursivo de manera incorrecta para que no converja en el caso base, puede ocasionar una recursividad infinita, con lo cual se agota la memoria en cierto punto. Este error es análogo al problema de un ciclo infinito en una solución iterativa (no recursiva).

Sección 15.4 Ejemplo de uso de recursividad: serie de Fibonacci

- La serie de Fibonacci empieza con 0 y 1, y tiene la propiedad de que cada número subsiguiente de Fibonacci es la suma de los dos anteriores.
- La proporción de números de Fibonacci sucesivos converge en un valor constante de 1.618…, un número al que se le denomina la proporción dorada, o media dorada.
- Algunas soluciones recursivas, como la de Fibonacci (que realiza dos llamadas por cada paso recursivo), producen una "explosión" de llamadas a métodos.

Sección 15.5 La recursividad y la pila de llamadas a métodos

- Una pila es una estructura de datos en la que sólo se pueden agregar o eliminar datos de la parte superior.
- Una pila es la analogía de un montón de platos. Cuando se coloca un plato en el montón, siempre se coloca en la parte superior (a esto se le conoce como meter el plato en la pila). De manera similar, cuando se quita un plato del montón, siempre se quita de la parte superior (a esto se le conoce como sacar el plato de la pila).
- Las pilas se conocen como estructuras de datos "último en entrar, primero en salir" (UEPS): el último elemento que se metió (insertó) en la pila es el primero que se saca (elimina) de ella.
- Las pilas tienen muchas aplicaciones interesantes. Por ejemplo, cuando un programa llama a un método, el método que se llamó debe saber cómo regresar al que lo llamó, por lo que se mete la dirección de retorno del método que hizo la llamada en la pila de ejecución del programa (a la que algunas veces se le conoce como la pila de llamadas a métodos).
- La pila de ejecución del programa contiene la memoria para las variables locales en cada invocación de un método, durante la ejecución de un programa. Estos datos, que se almacenan como una parte de la pila de ejecución del programa, se conocen como el registro de activación o marco de pila de la llamada al método.
- Si hay más llamadas a métodos recursivas o anidadas de las que pueden almacenarse en la pila de ejecución del programa, se produce un error conocido como desbordamiento de pila.

Sección 15.6 Comparación entre recursividad e iteración

- Tanto la iteración como la recursividad se basan en una instrucción de control: la iteración utiliza una instrucción de repetición, la recursividad una instrucción de selección.
- Tanto la iteración como la recursividad implican la repetición: la iteración utiliza de manera explícita una instrucción de repetición, mientras que la recursividad logra la repetición a través de llamadas repetidas a un método.
- La iteración y la recursividad implican una prueba de terminación: la iteración termina cuando falla la condición de continuación de ciclo, la recursividad cuando se reconoce un caso base.
- La iteración con repetición controlada por contador y la recursividad se acercan en forma gradual a la terminación: la iteración sigue modificando un contador, hasta que éste asume un valor que hace que falle la condición de continuación de ciclo, mientras que la recursividad sigue produciendo versiones cada vez más simples del problema original, hasta llegar al caso base.
- Tanto la iteración como la recursividad pueden ocurrir en forma infinita. Un ciclo infinito ocurre con la iteración si la prueba de continuación de ciclo nunca se vuelve falsa, mientras que la recursividad infinita ocurre si el paso recursivo no reduce el problema cada vez más, de una forma que converja en el caso base.
- La recursividad invoca el mecanismo en forma repetida, y en consecuencia a la sobrecarga producida por las llamadas al método.

- Cualquier problema que pueda resolverse en forma recursiva, se puede resolver también en forma iterativa.
- Por lo general se prefiere un método recursivo en vez de uno iterativo cuando el primero refleja el problema con más naturalidad, y produce un programa más fácil de comprender y de depurar.
- A menudo se puede implementar un método recursivo con pocas líneas de código, pero el método iterativo correspondiente podría requerir una gran cantidad de código. Otra razón por la que es más conveniente elegir una solución recursiva es que una solución iterativa podría no ser aparente.

Sección 15.8 Fractales
- Un fractal es una figura geométrica que se genera a partir de un patrón que se repite en forma recursiva, un número infinito de veces.
- Los fractales tienen una propiedad de auto-similitud: las subpartes son copias de tamaño reducido de toda la pieza.

Sección 15.9 "Vuelta atrás" recursiva (backtracking)
- Al uso de la recursividad para regresar a un punto de decisión anterior se le conoce como "vuelta atrás" recursiva. Si un conjunto de llamadas recursivas no produce como resultado una solución al problema, el programa retrocede hasta el punto de decisión anterior y toma una decisión distinta, lo cual a menudo produce otro conjunto de llamadas recursivas.

Terminología

caso base
converger en un caso base
Copo de nieve de Koch, fractal
Curva de Koch, fractal
desbordamiento de pila
evaluación recursiva
factorial
Fibonacci, serie de
Fractal
fractal auto-similar
fractal estrictamente auto-similar
llamada recursiva
marco de pila
media dorada
método recursivo
nivel de un fractal
nivel del fractal
Ocho Reinas, problema
orden del fractal
palíndromo
paso recursivo
pila
pila de ejecución del programa
pila de llamadas a métodos
profundidad del fractal
proporción dorada
prueba de terminación
recorrido del laberinto, problema
recursividad exhaustiva
recursividad indirecta
recursividad infinita
registro de activación
sobrecarga de ejecución
teoría de complejidad
torres de Hanoi, problema
último en entrar, primero en salir (UEPS), estructuras de datos
"vuelta atrás"
"vuelta atrás" recursiva

Ejercicios de autoevaluación

15.1 Conteste con *verdadero* o *falso* a cada una de las siguientes proposiciones; en caso de ser *falso*, explique por qué.
   a) Un método que se llama a sí mismo en forma indirecta no es un ejemplo de recursividad.
   b) La recursividad puede ser eficiente en la computación, debido a la reducción en el uso del espacio en memoria.
   c) Cuando se llama a un método recursivo para resolver un problema, en realidad es capaz de resolver sólo el (los) caso(s) más simple(s), o caso(s) base.
   d) Para que la recursividad sea factible, el paso recursivo en una solución recursiva debe asemejarse al problema original, pero debe ser una versión ligeramente más grande del mismo.

15.2 Para terminar la recursividad, se requiere un(a) ______________.
   a) paso recursivo
   b) instrucción `break`
   c) tipo de valor de retorno `void`
   d) caso base

15.3 La primera llamada para invocar a un método recursivo es ___________________.
a) no recursiva
b) recursiva
c) el paso recursivo
d) ninguna de las anteriores

15.4 Cada vez que se aplica el patrón de un fractal, se dice que el fractal está en un(a) nuevo(a) _______________.
a) anchura
b) altura
c) nivel
d) volumen

15.5 La iteración y la recursividad implican un(a) ___________________.
a) instrucción de repetición
b) prueba de terminación
c) variable contador
d) ninguna de las anteriores

15.6 Complete los siguientes enunciados:
a) La proporción de números de Fibonacci sucesivos converge en un valor constante de 1.618…, un número al que se le conoce como ___________________ o ___________________.
b) Sólo pueden agregarse o eliminarse datos de la ___________________ de la pila.
c) Las pilas se conocen como estructuras de datos ___________________; el último elemento que se metió (insertó) en la pila es el primer elemento que se saca (elimina) de ella.
d) La pila de ejecución del programa contiene la memoria para las variables locales en cada invocación de un método, durante la ejecución de un programa. Estos datos, que se almacenan como una parte de la pila de ejecución del programa, se conocen como el ___________________ o el ___________________ de llamadas a métodos.
e) Si hay más llamadas a métodos recursivas o anidadas de las que puedan almacenarse en la pila de ejecución del programa, se produce un error conocido como ___________________.
f) Por lo general, la iteración utiliza una instrucción de repetición, mientras que la recursividad comúnmente utiliza una instrucción ___________________.
g) Los fractales tienen una propiedad llamada ___________________; cuando se subdividen en partes, cada una de ellas es una copia de tamaño reducido de la pieza completa.
h) Las ___________________ de una cadena son todas las cadenas distintas que pueden crearse al reordenar los caracteres de la cadena original.
i) La pila de ejecución del programa se conoce también como la pila ___________________.

Respuestas a los ejercicios de autoevaluación

15.1 a) Falso. Un método que se llama a sí mismo en forma indirecta es un ejemplo de recursividad; en forma más específica, es un ejemplo de recursividad indirecta. b) Falso. La recursividad puede ser ineficiente en la computación debido a las múltiples llamadas a un método y el uso del espacio de memoria. c) Verdadero. d) Falso. Para que la recursividad sea factible, el paso recursivo en una solución recursiva debe asemejarse al problema original, pero debe ser una versión ligeramente *más pequeña* del mismo.

15.2 d

15.3 a

15.4 c

15.5 b

15.6 a) proporción dorada, media dorada. b) parte superior. c) último en entrar, primero en salir (UEPS). d) registro de activación, marco de pila. e) desbordamiento de pila. f) de selección. g) auto-similitud. h) permutaciones. i) de llamadas a métodos.

Ejercicios

15.7 ¿Qué hace el siguiente código?

```
public int misterio( int a, int b )
{
   if ( b == 1 )
      return a;
   else
      return a + misterio( a, b - 1 );
} // fin del método misterio
```

15.8 Busque el(los) error(es) en el siguiente método recursivo, y explique cómo corregirlo(s). Este método debe encontrar la suma de los valores de 0 a `n`.

```
public int suma( int n )
{
   if ( n == 0 )
      return 0;
   else
      return n + suma( n );
} // fin del método suma
```

15.9 *(Método `potencia` recursivo)* Escriba un método recursivo llamado `potencia( base, exponente )` que, cuando sea llamado, devuelva

$$base^{exponente}$$

Por ejemplo, `potencia( 3, 4 )` = 3 * 3 *3 * 3. Suponga que `exponente` es un entero mayor o igual que `1`. [*Sugerencia:* el paso recursivo debe utilizar la relación

$$base^{exponente} = base \cdot base^{exponente - 1}$$

y la condición de terminación ocurre cuando `exponente` es igual a `1`, ya que

$$base^{1} = base$$

Incorpore este método en un programa que permita al usuario introducir la `base` y el `exponente`].

15.10 *(Visualización de la recursividad)* Es interesante observar la recursividad "en acción". Modifique el método factorial de la figura 15.3 para imprimir su variable local y su parámetro de llamada recursiva. Para cada llamada recursiva, muestre los resultados en una línea separada y agregue un nivel de sangría. Haga su máximo esfuerzo por hacer que los resultados sean claros, interesantes y significativos. Su meta aquí es diseñar e implementar un formato de salida que facilite la comprensión de la recursividad. Tal vez desee agregar ciertas capacidades de visualización a otros ejemplos y ejercicios recursivos a lo largo de este libro.

15.11 *(Máximo común divisor)* El máximo común divisor de los enteros `x` y `y` es el entero más grande que se puede dividir entre `x` y `y` de manera uniforme. Escriba un método recursivo llamado `mcd`, que devuelva el máximo común divisor de `x` y `y`. El `mcd` de `x` y `y` se define, mediante la recursividad, de la siguiente manera: si `y` es igual a `0`, entonces `mcd( x, y )` es `x`; en caso contrario, `mcd( x, y )` es `mcd( y, x % y )`, en donde `%` es el operador residuo. Use este método para sustituir el que escribió en la aplicación del ejercicio 6.27.

15.12 ¿Qué hace el siguiente programa?

```
// Ejercicio 15.12 Solución: ClaseMisteriosa.java

public class ClaseMisteriosa
{
   public int misterio( int arreglo2[], int tamanio )
   {
      if ( tamanio == 1 )
         return arreglo2[ 0 ];
```

```
      else
         return arreglo2[ tamanio - 1 ] + misterio( arreglo2, tamanio - 1 );
   } // fin del método misterio
} // fin de la clase ClaseMisteriosa
```

```
// Ejercicio 15.12 Solución: PruebaMisteriosa.java

public class PruebaMisteriosa
{
   public static void main( String args[] )
   {
      ClaseMisteriosa objetoMisterioso = new ClaseMisteriosa();

      int arreglo[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

      int resultado = objetoMisterioso.misterio( arreglo, arreglo.length );

      System.out.printf( "El resultado es: %d\n", resultado );
   } // fin del método main
} // fin de la clase PruebaMisteriosa
```

15.13 ¿Qué hace el siguiente programa?

```
// Ejercicio 15.13 Solución: UnaClase.java

public class UnaClase
{
   public String unMetodo(
      int arreglo2[], int x, String salida )
   {
      if ( x < arreglo2.length )
         return String.format(
            "%s%d ", unMetodo( arreglo2, x + 1 ), arreglo2[ x ] );
      else
         return "";
   } // fin del método unMetodo
} // fin de la clase UnaClase
```

```
// Ejercicio 15.13 Solución: PruebaUnaClase.java

public class PruebaUnaClase
{
   public static void main( String args[] )
   {
      UnaClase objetoUnaClase = new UnaClase();

      int arreglo[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

      String resultados =
         objetoUnaClase.unMetodo( arreglo, 0 );

      System.out.println( resultados );
   } // fin de main
} // fin de la clase PruebaUnaClase
```

15.14 *(Palíndromos)* Un palíndromo es una cadena que se escribe de la misma forma tanto al derecho como al revés. Algunos ejemplos de palíndromos son "radar", "reconocer" y (si se ignoran los espacios) "anita lava la tina". Escriba un método recursivo llamado `probarPalindromo`, que devuelva el valor `boolean true` si la cadena almacenada en el arreglo es un palíndromo, y `false` en caso contrario. El método debe ignorar espacios y puntuación en la cadena.

15.15 *(Ocho reinas)* Un buen acertijo para los fanáticos del ajedrez es el problema de las Ocho reinas, que se describe a continuación: ¿es posible colocar ocho reinas en un tablero de ajedrez vacío, de forma que ninguna reina "ataque" a otra (es decir, que no haya dos reinas en la misma fila, en la misma columna o a lo largo de la misma diagonal)? Por ejemplo, si se coloca una reina en la esquina superior izquierda del tablero, no pueden colocarse otras reinas en ninguna de las posiciones marcadas que se muestran en la figura 15.23. Resuelva el problema mediante el uso de recursividad. [*Sugerencia:* su solución debe empezar con la primera columna y buscar una ubicación en esa columna, en donde pueda colocarse una reina; al principio, coloque la reina en la primera fila. Después, la solución debe buscar en forma recursiva el resto de las columnas. En las primeras columnas, habrá varias ubicaciones en donde pueda colocarse una reina. Tome la primera posición disponible. Si se llega a una columna sin que haya una posible ubicación para una reina, el programa deberá regresar a la columna anterior y desplazar la reina que está en esa columna hacia una nueva fila. Este proceso continuo de retroceder y probar nuevas alternativas es un ejemplo de la "vuelta atrás" recursiva].

15.16 *(Imprimir un arreglo)* Escriba un método recursivo llamado `imprimirArreglo`, que muestre todos los elementos en un arreglo de enteros, separados por espacios.

15.17 *(Imprimir un arreglo al revés)* Escriba un método recursivo llamado `cadenaInversa`, que reciba un arreglo de caracteres que contenga una cadena como argumento, y que la imprima al revés. [*Sugerencia:* use el método `String` llamado `toCharArray`, el cual no recibe argumentos, para obtener un arreglo `char` que contenga los caracteres en el objeto `String`.]

15.18 *(Buscar el valor mínimo en un arreglo)* Escriba un método recursivo llamado `minimoRecursivo`, que determine el elemento más pequeño en un arreglo de enteros. Este método deberá regresar cuando reciba un arreglo de un elemento.

15.19 *(Fractales)* Repita el patrón del fractal de la sección 15.8 para formar una estrella. Empiece con cinco líneas en vez de una, en donde cada línea es un pico distinto de la estrella. Aplique el patrón del "fractal Lo" a cada pico de la estrella.

15.20 *(Recorrido de un laberinto mediante el uso de la "vuelta atrás" recursiva)* La cuadrícula que contiene caracteres `#` y puntos (`.`) en la figura 15.24 es una representación de un laberinto mediante un arreglo bidimensional. Los caracteres `#` representan las paredes del laberinto, y los puntos representan las ubicaciones en las posibles rutas a través del laberinto. Sólo pueden realizarse movimientos hacia una ubicación en el arreglo que contenga un punto.

Escriba un método recursivo (`recorridoLaberinto`) para avanzar a través de laberintos como el de la figura 15.24. El método debe recibir como argumentos un arreglo de caracteres de 12 por 12 que representa el laberinto, y la posición actual en el laberinto (la primera vez que se llama a este método, la posición actual debe ser el punto de entrada del laberinto). A medida que `recorridoLaberinto` trate de localizar la salida, debe colocar el carácter `x` en cada posición en la ruta. Hay un algoritmo simple para avanzar a través de un laberinto, que garantiza encontrar la salida (suponiendo que haya una). Si no hay salida, llegaremos a la posición inicial de nuevo. El algoritmo es el siguiente: partiendo de la posición actual en el laberinto, trate de avanzar un espacio en cualquiera de las posibles direcciones (abajo, derecha, arriba o izquierda). Si es posible avanzar por lo menos en una dirección, llame a `recorridoLaberinto` en forma recursiva, pasándole la nueva posición en el laberinto como la posición actual. Si no es posible avanzar en ninguna dirección, "retroceda" a una posición anterior en el laberinto y pruebe una nueva dirección para esa posición. Programe

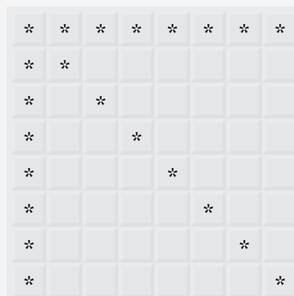


Figura 15.23 | Eliminación de posiciones al colocar una reina en la esquina superior izquierda de un tablero de ajedrez.

el método para que muestre el laberinto después de cada movimiento, de manera que el usuario pueda observar a la hora de que se resuelva el laberinto. La salida final del laberinto deberá mostrar sólo la ruta necesaria para resolverlo; si al ir en una dirección específica se llega a un punto sin salida, no se deben mostrar las `x` que avancen en esa dirección. [*Sugerencia:* para mostrar sólo la ruta final, tal vez sea útil marcar las posiciones que resulten en un punto sin salida con otro carácter (como `'0'`)].

15.21 *(Generación de laberintos al azar)* Escriba un método llamado `generadorLaberintos`, que reciba como argumento un arreglo bidimensional de 12 por 12 caracteres, y que produzca un laberinto al azar. Este método también deberá proporcionar las posiciones inicial y final del laberinto. Pruebe su método `recorridoLaberinto` del ejercicio 15.20, usando varios laberintos generados al azar.

15.22 *(Laberintos de cualquier tamaño)* Generalice los métodos `recorridoLaberinto` y `generadorLaberintos` de los ejercicios 15.20 y 15.21 para procesar laberintos de cualquier anchura y altura.

15.23 *(Tiempo para calcular números de Fibonacci)* Mejore el programa de Fibonacci de la figura 15.5, de manera que calcule el monto de tiempo aproximado requerido para realizar el cálculo, y el número de llamadas realizadas al método recursivo. Para este fin, llame al método `static` de `System` llamado `currentTimeMillis`, el cual no recibe argumento y devuelve el tiempo actual de la computadora en milisegundos. Llame a este método dos veces; una antes y la otra después de la llamada a `fibonacci`. Guarde cada valor y calcule la diferencia en los tiempos, para determinar cuántos milisegundos se requirieron para realizar el cálculo. Después, agregue una variable a la clase `CalculoFibonacci`, y utilice esta variable para determinar el número de llamadas realizadas al método `fibonacci`. Muestre sus resultados.

```
# # # # # # # # # # # #
# . . . # . . . . . . #
. . # . # . # # # # . #
# # # . # . . . . # . #
# . . . . # # # . # . .
# # # # . # . # . # . #
# . . # . # . # . # . #
# # . # . # . # . # . #
# . . . . . . . . # . #
# # # # # # . # # # . #
# . . . . . . # . . . #
# # # # # # # # # # # #
```

Figura 15.24 | Representación de un laberinto mediante un arreglo bidimensional.

16

Búsqueda y ordenamiento



Con sollozos y lágrimas él sorteó
Los de mayor tamaño…

—Lewis Carroll

Intenta el final, y nunca dejes lugar a dudas;
No hay nada tan difícil que no pueda averiguarse mediante la búsqueda.

—Robert Herrick

Está bloqueado en mi memoria,
Y tú deberás guardar la llave.

—William Shakespeare

Una ley inmutable en los negocios es que las palabras son palabras, las explicaciones son explicaciones, las promesas son promesas; pero sólo el desempeño es la realidad.

—Harold S. Green

OBJETIVOS

En este capítulo aprenderá a:

- Buscar un valor dado en un arreglo, usando la búsqueda lineal y la búsqueda binaria.
- Ordenar arreglos, usando los algoritmos iterativos de ordenamiento por selección y por inserción.
- Ordenar arreglos, usando el algoritmo recursivo de ordenamiento por combinación.
- Determinar la eficiencia de los algoritmos de búsqueda y ordenamiento.
- Usar invariantes de ciclo para ayudar a asegurar que los programas sean correctos.

Plan general

16.1 Introducción

La **búsqueda** de datos implica el determinar si un valor (conocido como la **clave de búsqueda**) está presente en los datos y, de ser así, hay que encontrar su ubicación. Dos algoritmos populares de búsqueda son la búsqueda lineal simple y la búsqueda binaria, que es más rápida pero a la vez más compleja. El **ordenamiento** coloca los datos en orden ascendente o descendente, con base en una o más **claves de ordenamiento**. Una lista de nombres se podría ordenar en forma alfabética, las cuentas bancarias podrían ordenarse por número de cuenta, los registros de nóminas de empleados podrían ordenarse por número de seguro social, etcétera. En este capítulo se presentan dos algoritmos de ordenamiento simples, el ordenamiento por selección y el ordenamiento por inserción, junto con el ordenamiento por combinación, que es más eficiente pero también más complejo. En la figura 16.1 se sintetizan los algoritmos de búsqueda y ordenamiento que veremos en los ejemplos y ejercicios de este libro.

Capítulo	Algoritmo	Ubicación
Algoritmos de búsqueda:		
16	Búsqueda lineal.	Sección 16.2.1.
	Búsqueda binaria.	Sección 16.2.2.
	Búsqueda lineal recursiva.	Ejercicio 16.8.
	Búsqueda binaria recursiva.	Ejercicio 16.9.
17	Búsqueda lineal en un objeto `Lista`.	Ejercicio 17.21.
	Búsqueda en un árbol binario.	Ejercicio 17.23.
19	El método `binarySearch` de `Collections`.	Figura 19.14.
Algoritmos de ordenamiento:		
16	Ordenamiento por selección.	Sección 16.3.1.
	Ordenamiento por inserción.	Sección 16.3.2.
	Ordenamiento por combinación.	Sección 16.3.3.
	Ordenamiento de burbuja.	Ejercicios 16.3 y 16.4.
	Ordenamiento de cubeta.	Ejercicio 16.7.
	Ordenamiento rápido (quicksort) recursivo.	Ejercicio 16.10.
17	Ordenamiento con árboles binarios.	Sección 17.9.
19	El método `sort` de `Collections`.	Figuras 19.8 a 19.11.
	Colección `SortedSet`.	Figura 19.19.

Figura 16.1 | Los algoritmos de búsqueda y ordenamiento de este libro.

16.2 Algoritmos de búsqueda

Buscar un número telefónico, buscar un sitio Web a través de un motor de búsqueda y comprobar la definición de una palabra en un diccionario son acciones que implican buscar entre grandes cantidades de datos. En las siguientes dos secciones hablaremos sobre dos algoritmos de búsqueda comunes: uno que es fácil de programar, pero relativamente ineficiente, y uno que es relativamente eficiente pero más complejo y difícil de programar.

16.2.1 Búsqueda lineal

El **algoritmo de búsqueda lineal** busca por cada elemento de un arreglo en forma secuencial. Si la clave de búsqueda no coincide con un elemento en el arreglo, el algoritmo evalúa cada elemento y, cuando se llega al final del arreglo, informa al usuario que la clave de búsqueda no está presente. Si la clave de búsqueda se encuentra en el arreglo, el algoritmo evalúa cada elemento hasta encontrar uno que coincida con la clave de búsqueda y devuelve el índice de ese elemento.

Como ejemplo, considere un arreglo que contiene los siguientes valores:

```
34   56   2   10   77   51   93   30   5   52
```

y un programa que busca el número 51. Usando el algoritmo de búsqueda lineal, el programa primero comprueba si el 34 coincide con la clave de búsqueda. Si no es así, el algoritmo comprueba si 56 coincide con la clave de búsqueda. El programa continúa recorriendo el arreglo en forma secuencial, y evalúa el 2, luego el 10, después el 77. Cuando el programa evalúa el número 51, que coincide con la clave de búsqueda, devuelve el índice 5, que está en la posición del 51 en el arreglo. Si, después de comprobar cada elemento del arreglo, el programa determina que la clave de búsqueda no coincide con ningún elemento del arreglo, el programa devuelve un valor centinela (por ejemplo, `-1`).

En la figura 16.2 se declara la clase `ArregloLineal`. Esta clase tiene dos variables de instancia `private`: un arreglo de valores `int` llamado `datos`, y un objeto `static Random` para llenar el arreglo con valores `int` generados al azar. Cuando se crea una instancia de un objeto de la clase `ArregloLineal`, el constructor (líneas 12 a 19) crea e inicializa el arreglo `datos` con valores `int` aleatorios en el rango de `10 a 99`. Si hay valores duplicados en el arreglo, la búsqueda lineal devuelve el índice del primer elemento en el arreglo que coincide con la clave de búsqueda.

```
1  // Fig. 16.2: ArregloLineal.java
2  // Clase que contiene un arreglo de enteros aleatorios y un método
3  // que busca en ese arreglo, en forma secuencial.
4  import java.util.Random;
5
6  public class ArregloLineal
7  {
8     private int[] datos; // arreglo de valores
9     private static Random generador = new Random();
10
11    // crea un arreglo de un tamaño dado, y lo rellena con enteros aleatorios
12    public ArregloLineal( int tamanio )
13    {
14       datos = new int[ tamanio ]; // crea un espacio para el arreglo
15
16       // llena el arreglo con valores int aleatorios, en el rango de 10 a 99
17       for ( int i = 0; i < tamanio; i++ )
18          datos[ i ] = 10 + generador.nextInt( 90 );
19    } // fin del constructor de ArregloLineal
20
21    // realiza una búsqueda lineal en los datos
22    public int busquedaLineal( int claveBusqueda )
23    {
24       // itera a través del arreglo en forma secuencial
```

Figura 16.2 | La clase `ArregloLineal`. (Parte 1 de 2).

```
25          for ( int indice = 0; indice < datos.length; indice++ )
26             if ( datos[ indice ] == claveBusqueda )
27                return indice; // devuelve el índice del entero
28
29          return -1; // no se encontró el entero
30       } // fin del método busquedaLineal
31
32       // método para imprimir los valores del arreglo
33       public String toString()
34       {
35          StringBuilder temporal = new StringBuilder();
36
37          // itera a través del arreglo
38          for ( int elemento : datos )
39             temporal.append( elemento + " " );
40
41          temporal.append( "\n" ); // agrega el carácter de nueva línea
42          return temporal.toString();
43       } // fin del método toString
44    } // fin de la clase ArregloLineal
```

Figura 16.2 | La clase `ArregloLineal`. (Parte 2 de 2).

En las líneas 22 a 30 se realiza la búsqueda lineal. La clave de búsqueda se pasa al parámetro `claveBusqueda`. En las líneas 25 a 27 se itera a través de los elementos en el arreglo. En la línea 26 se compara cada elemento en el arreglo con `claveBusqueda`. Si los valores son iguales, en la línea 27 se devuelve el índice del elemento. Si el ciclo termina sin encontrar el valor, en la línea 29 se devuelve `-1`. En las líneas 33 a 43 se declara el método `toString`, que devuelve una representación `String` del arreglo para imprimirlo.

En la figura 16.3 se crea un objeto `ArregloLineal`, el cual contiene un arreglo de `10` valores `int` (línea 16) y permite al usuario buscar elementos específicos en el arreglo. En las líneas 20 a 22 se pide al usuario la clave de búsqueda y se almacena en `enteroBusqueda`. Después, en las líneas 25 a 41 se itera hasta que el `enteroBusqueda` sea igual a `-1`. El arreglo contiene valores `int` de `10 a 99` (línea 18 de la figura 16.2). En la línea 28 se llama al método `busquedaLineal` para determinar si `enteroBusqueda` está en el arreglo. Si no lo está, `busquedaLineal` devuelve `-1` y el programa notifica al usuario (líneas 31 y 32). Si `enteroBusqueda` está en el arreglo, `busquedaLineal` devuelve la posición del elemento, que el programa muestra en las líneas 34 y 35. En las líneas 38 a 40 se obtiene el siguiente entero del usuario.

```
1    // Fig. 16.3: PruebaBusquedaLineal.java
2    // Busca un elemento en el arreglo, en forma secuencial.
3    import java.util.Scanner;
4
5    public class PruebaBusquedaLineal
6    {
7       public static void main( String args[] )
8       {
9          // crea objeto Scanner para los datos de entrada
10         Scanner entrada = new Scanner( System.in );
11
12         int enteroBusqueda; // clave de búsqueda
13         int posicion; // ubicación de la clave de búsqueda en el arreglo
14
15         // crea el arreglo y lo muestra en pantalla
16         ArregloLineal arregloBusqueda = new ArregloLineal( 10 );
17         System.out.println( arregloBusqueda ); // imprime el arreglo
```

Figura 16.3 | La clase `PruebaBusquedaLineal`. (Parte 1 de 2).

```

      // obtiene la entrada del usuario
      System.out.print(
         "Escriba un valor entero (-1 para terminar): " );
      enteroBusqueda = entrada.nextInt(); // lee el primer entero del usuario

      // recibe en forma repetida un entero como entrada; -1 termina el programa
      while ( enteroBusqueda != -1 )
      {
         // realiza una búsqueda lineal
         posicion = arregloBusqueda.busquedaLineal( enteroBusqueda );

         if ( posicion == -1 ) // no se encontró el entero
            System.out.println( "El entero " + enteroBusqueda +
               " no se encontro.\n" );
         else // se encontró el entero
            System.out.println( "El entero " + enteroBusqueda +
               " se encontro en la posicion " + posicion + ".\n" );

         // obtiene la entrada del usuario
         System.out.print(
            "Escriba un valor entero (-1 para terminar): " );
         enteroBusqueda = entrada.nextInt(); // lee el siguiente entero del usuario
      } // fin de while
   } // fin de main
} // fin de la clase PruebaBusquedaLineal
```

```
59 13 96 85 68 23 64 49 58 79

Escriba un valor entero (-1 para terminar): 68
El entero 68 se encontro en la posicion 4.

Escriba un valor entero (-1 para terminar): 49
El entero 49 se encontro en la posicion 7.

Escriba un valor entero (-1 para terminar): 33
El entero 33 no se encontro.

Escriba un valor entero (-1 para terminar): -1
```

Figura 16.3 | La clase `PruebaBusquedaLineal`. (Parte 2 de 2).

Eficiencia de la búsqueda lineal

Todos los algoritmos de búsqueda logran el mismo objetivo: encontrar un elemento que coincida con una clave de búsqueda dada, si es que existe dicho elemento. Sin embargo, hay varias cosas que diferencian a un algoritmo de otro. La principal diferencia es la cantidad de esfuerzo que requieren para completar la búsqueda. Una forma de describir este esfuerzo es mediante la **notación Big O**, la cual indica el tiempo de ejecución para el peor caso de un algoritmo; es decir, qué tan duro tendrá que trabajar un algoritmo para resolver un problema. En los algoritmos de búsqueda y ordenamiento, esto depende específicamente de cuántos elementos de datos haya.

Suponga que un algoritmo está diseñado para evaluar si el primer elemento de un arreglo es igual al segundo elemento. Si el arreglo tiene 10 elementos, este algoritmo requiere una comparación. Si el arreglo tiene 1000 elementos, sigue requiriendo una comparación. De hecho, el algoritmo es completamente independiente del número de elementos en el arreglo. Se dice que este algoritmo tiene un **tiempo de ejecución constante**, el cual se representa en la notación Big O como ***O*(1)**. Un algoritmo que es *O*(1) no necesariamente requiere sólo de una comparación. *O*(1) sólo significa que el número de comparaciones es *constante*; no crece a medida que aumenta el tamaño del arreglo. Un algoritmo que evalúa si el primer elemento de un arreglo es igual a los siguientes tres elementos sigue siendo *O*(1), aun cuando requiera tres comparaciones.

Un algoritmo que evalúa si el primer elemento de un arreglo es igual a *cualquiera* de los demás elementos del arreglo requerirá cuando menos de $n - 1$ comparaciones, en donde n es el número de elementos en el arreglo. Si el arreglo tiene 10 elementos, este algoritmo requiere hasta nueve comparaciones. Si el arreglo tiene 1000 elementos, requiere hasta 999 comparaciones. A medida que n aumenta en tamaño, la parte de la expresión correspondiente a la n "domina", y si le restamos uno no hay consecuencias. Big O está diseñado para resaltar estos términos dominantes e ignorar los términos que pierden importancia, a medida que n crece. Por esta razón, se dice que un algoritmo que requiere un total de $n - 1$ comparaciones (como el que describimos antes) es ***O(n)***. Se considera que un algoritmo *O(n)* tiene un **tiempo de ejecución lineal**. A menudo, *O(n)* significa "en el orden de n", o dicho en forma más simple, "orden n".

Ahora, suponga que tiene un algoritmo que evalúa si *cualquier* elemento de un arreglo se duplica en cualquier otra parte del mismo. El primer elemento debe compararse con todos los demás elementos del arreglo. El segundo elemento debe compararse con todos los demás elementos, excepto con el primero (ya se comparó con éste). El tercer elemento debe compararse con todos los elementos, excepto los primeros dos. Al final, este algoritmo terminará realizando $(n - 1) + (n - 2) + \ldots + 2 + 1$, o $n^2/2 - n/2$ comparaciones. A medida que n aumenta, el término n^2 domina y el término n se vuelve inconsecuente. De nuevo, la notación Big O resalta el término n^2, dejando a $n^2/2$. Pero como veremos pronto, los factores constantes se omiten en la notación Big O.

Big O se enfoca en la forma en que aumenta el tiempo de ejecución de un algoritmo, en relación con el número de elementos procesados. Suponga que un algoritmo requiere n^2 comparaciones. Con cuatro elementos, el algoritmo requiere 16 comparaciones; con ocho elementos, 64 comparaciones. Con este algoritmo, al duplicar el número de elementos se cuadruplica el número de comparaciones. Considere un algoritmo similar que requiere $n^2/2$ comparaciones. Con cuatro elementos, el algoritmo requiere ocho comparaciones; con ocho elementos, 32 comparaciones. De nuevo, al duplicar el número de elementos se cuadruplica el número de comparaciones. Ambos de estos elementos aumentan como el cuadrado de n, por lo que Big O ignora la constante y ambos algoritmos se consideran como ***O(n^2)***, lo cual se conoce como **tiempo de ejecución cuadrático** y se pronuncia como "en el orden de n al cuadrado", o dicho en forma más simple, "orden n al cuadrado".

Cuando n es pequeña, los algoritmos *O(n^2)* (que se ejecutan en las computadoras personales de la actualidad, con miles de millones de operaciones por segundo) no afectan el rendimiento en forma considerable. Pero a medida que n aumenta, se empieza a notar la reducción en el rendimiento. Un algoritmo *O(n^2)* que se ejecuta en un arreglo de un millón de elementos requeriría un billón de "operaciones" (en donde cada una requeriría en realidad varias instrucciones de máquina para ejecutarse). Esto podría requerir varias horas para ejecutarse. Un arreglo de mil millones de elementos requeriría un trillón de operaciones, ¡un número tan grande que el algoritmo tardaría décadas! Por desgracia, los algoritmos *O(n^2)* son fáciles de escribir, como veremos en este capítulo. También veremos algoritmos con medidas de Big O más favorables. Estos algoritmos eficientes comúnmente requieren un poco más de astucia y trabajo para crearlos, pero su rendimiento superior bien vale la pena el esfuerzo adicional, en especial a medida que n aumenta y los algoritmos se combinan en programas más grandes.

El algoritmo de búsqueda lineal se ejecuta en un tiempo *O(n)*. El peor caso en este algoritmo es que se debe comprobar cada elemento para determinar si el elemento que se busca existe en el arreglo. Si el tamaño del arreglo se duplica, el número de comparaciones que el algoritmo debe realizar también se duplica. Observe que la búsqueda lineal puede proporcionar un rendimiento sorprendente, si el elemento que coincide con la clave de búsqueda se encuentra en (o cerca de) la parte frontal del arreglo. Pero buscamos algoritmos que tengan un buen desempeño, en promedio, en todas las búsquedas, incluyendo aquellas en las que el elemento que coincide con la clave de búsqueda se encuentra cerca del final del arreglo.

La búsqueda lineal es el algoritmo de búsqueda más fácil de programar, pero puede ser lento si se le compara con otros algoritmos de búsqueda. Si un programa necesita realizar muchas búsquedas en arreglos grandes, puede ser mejor implementar un algoritmo más eficiente, como la búsqueda binaria, el cual presentaremos en la siguiente sección.



Tip de rendimiento 16.1

Algunas veces los algoritmos más simples tienen un desempeño pobre. Su virtud es que son fáciles de programar, probar y depurar. En ocasiones se requieren algoritmos más complejos para obtener el máximo rendimiento.

16.2.2 Búsqueda binaria

El **algoritmo de búsqueda binaria** es más eficiente que el algoritmo de búsqueda lineal, pero requiere que el arreglo se ordene. La primera iteración de este algoritmo evalúa el elemento medio del arreglo. Si éste coincide con

la clave de búsqueda, el algoritmo termina. Suponiendo que el arreglo se ordene en forma ascendente, entonces si la clave de búsqueda es menor que el elemento de en medio, no puede coincidir con ningún elemento en la segunda mitad del arreglo, y el algoritmo continúa sólo con la primera mitad (es decir, el primer elemento hasta, pero sin incluir, el elemento de en medio). Si la clave de búsqueda es mayor que el elemento de en medio, no puede coincidir con ninguno de los elementos de la primera mitad del arreglo, y el algoritmo continúa sólo con la segunda mitad del arreglo (es decir, desde el elemento después del elemento de en medio, hasta el último elemento). Cada iteración evalúa el valor medio de la porción restante del arreglo. Si la clave de búsqueda no coincide con el elemento, el algoritmo elimina la mitad de los elementos restantes. Para terminar, el algoritmo encuentra un elemento que coincide con la clave de búsqueda o reduce el subarreglo hasta un tamaño de cero.

Como ejemplo, considere el siguiente arreglo ordenado de 15 elementos:

```
2   3   5   10   27   30   34   51   65   77   81  82   93   99
```

y una clave de búsqueda de 65. Un programa que implemente el algoritmo de búsqueda binaria primero comprobaría si el 51 es la clave de búsqueda (ya que 51 es el elemento de en medio del arreglo). La clave de búsqueda (65) es mayor que 51, por lo que este número se descarta junto con la primera mitad del arreglo (todos los elementos menores que 51). A continuación, el algoritmo comprueba si 81 (el elemento de en medio del resto del arreglo) coincide con la clave de búsqueda. La clave de búsqueda (65) es menor que 81, por lo que se descarta este número junto con los elementos mayores de 81. Después de sólo dos pruebas, el algoritmo ha reducido el número de valores a comprobar a tres (56, 65 y 77). Después el algoritmo comprueba el 65 (que coincide indudablemente con la clave de búsqueda), y devuelve el índice del elemento del arreglo que contiene el 65. Este algoritmo sólo requirió tres comparaciones para determinar si la clave de búsqueda coincidió con un elemento del arreglo. Un algoritmo de búsqueda lineal hubiera requerido 10 comparaciones. [*Nota:* en este ejemplo hemos optado por usar un arreglo con 15 elementos, para que siempre haya un elemento obvio en medio del arreglo. Con un número par de elementos, la parte media del arreglo se encuentra entre dos elementos. Implementamos el algoritmo para elegir el menor de esos dos elementos].

La figura 16.4 declara la clase `ArregloBinario`. Esta clase es similar a `ArregloLineal`: tiene dos variables de instancia `private`, un constructor, un método de búsqueda (`busquedaBinaria`), un método `elementosRestantes` y un método `toString`. En las líneas 13 a 22 se declara el constructor. Una vez que se inicializa el arreglo con valores `int` aleatorios de `10 a 99` (líneas 18 y 19), en la línea 21 se hace una llamada al método `Arrays.sort` en el arreglo `datos`. El método **`sort`** es un método `static` de la clase `Arrays`, que ordena los elementos en un arreglo en orden ascendente de manera predeterminada; una versión sobrecargada de este método nos permite cambiar la forma de ordenar los datos. Recuerde que el algoritmo de búsqueda binaria sólo funciona en un arreglo ordenado.

```
1  // Fig. 16.4: ArregloBinario.java
2  // Clase que contiene un arreglo de enteros aleatorios y un método
3  // que utiliza la búsqueda binaria para encontrar un entero.
4  import java.util.Random;
5  import java.util.Arrays;
6
7  public class ArregloBinario
8  {
9     private int[] datos; // arreglo de valores
10    private static Random generador = new Random();
11
12    // crea un arreglo de un tamaño dado y lo llena con enteros aleatorios
13    public ArregloBinario( int tamanio )
14    {
15       datos = new int[ tamanio ]; // crea espacio para el arreglo
16
17       // llena el arreglo con enteros aleatorios en el rango de 10 a 99
18       for ( int i = 0; i < tamanio; i++ )
19          datos[ i ] = 10 + generador.nextInt( 90 );
20
21       Arrays.sort( datos );
```

Figura 16.4 | La clase `ArregloBinario`. (Parte 1 de 2).

```java
   } // fin del constructor de ArregloBinario

   // realiza una búsqueda binaria en los datos
   public int busquedaBinaria( int elementoBusqueda )
   {
      int inferior = 0; // extremo inferior del área de búsqueda
      int superior = datos.length - 1; // extremo superior del área de búsqueda
      int medio = ( inferior + superior + 1 ) / 2; // elemento medio
      int ubicacion = -1; // devuelve el valor; -1 si no lo encontró

      do // ciclo para buscar un elemento
      {
         // imprime el resto de los elementos del arreglo
         System.out.print( elementosRestantes( inferior, superior ) );

         // imprime espacios para alineación
         for ( int i = 0; i < medio; i++ )
            System.out.print( "   " );
         System.out.println( " * " ); // indica el elemento medio actual

         // si el elemento se encuentra en medio
         if ( elementoBusqueda == datos[ medio ] )
            ubicacion = medio; // la ubicación es el elemento medio actual

         // el elemento medio es demasiado alto
         else if ( elementoBusqueda < datos[ medio ] )
            superior = medio - 1; // elimina la mitad superior
         else // el elemento medio es demasiado bajo
            inferior = medio + 1; // elimina la mitad inferior

         medio = ( inferior + superior + 1 ) / 2; // recalcula el elemento medio
      } while ( ( inferior <= superior ) && ( ubicacion == -1 ) );

      return ubicacion; // devuelve la ubicación de la clave de búsqueda
   } // fin del método busquedaBinaria

   // método para imprimir ciertos valores en el arreglo
   public String elementosRestantes( int inferior, int superior )
   {
      StringBuilder temporal = new StringBuilder();

      // imprime espacios para alineación
      for ( int i = 0; i < inferior; i++ )
         temporal.append( "     " );

      // imprime los elementos que quedan en el arreglo
      for ( int i = inferior; i <= superior; i++ )
         temporal.append( datos[ i ] + " " );

      temporal.append( "\n" );
      return temporal.toString();
   } // fin del método elementosRestantes

   // método para imprimir los valores en el arreglo
   public String toString()
   {
      return elementosRestantes( 0, datos.length - 1 );
   } // fin del método toString
} // fin de la clase ArregloBinario
```

Figura 16.4 | La clase `ArregloBinario`. (Parte 2 de 2).

En las líneas 25 a 56 se declara el método `busquedaBinaria`. La clave de búsqueda se pasa al parámetro `elementoBusqueda` (línea 25). En las líneas 27 a 29 se calcula el índice del extremo `inferior`, el índice del extremo `superior` y el índice `medio` de la porción del arreglo en la que el programa está buscando actualmente. Al principio del método, el extremo `inferior` es `0`, el extremo `superior` es la longitud del arreglo menos `1`, y `medio` es el promedio de estos dos valores. En la línea 30 se inicializa la `ubicacion` del elemento en `-1`; el valor que se devolverá si no se encuentra el elemento. En las líneas 32 a 53 se itera hasta que `inferior` sea mayor que `superior` (esto ocurre cuando no se encuentra el elemento), o cuando `ubicacion` no sea igual a `-1` (lo cual indica que se encontró la clave de búsqueda). En la línea 43 se evalúa si el valor en el elemento `medio` es igual a `elementoBusqueda`. Si esto es `true`, en la línea 44 se asigna `medio` a `ubicacion`. Después el ciclo termina y `ubicacion` se devuelve al método que hizo la llamada. Cada iteración del ciclo evalúa un solo valor (línea 43) y elimina la mitad del resto de los valores en el arreglo (línea 48 o 50).

En las líneas 26 a 44 de la figura 16.5 se itera hasta que el usuario escriba `-1`. Para cada uno de los otros números que escriba el usuario, el programa realiza una búsqueda binaria en los datos para determinar si coinciden con un elemento en el arreglo. La primera línea de salida de este programa es el arreglo de valores `int`, en orden ascendente. Cuando el usuario indica al programa que busque el número `23`, el programa primero evalúa el elemento medio, que es `42` (según lo indicado por el símbolo `*`). La clave de búsqueda es menor que `42`, por lo que el programa elimina la segunda mitad del arreglo y evalúa el elemento medio de la primera mitad. La clave de búsqueda es menor que `34`, por lo que el programa elimina la segunda mitad del arreglo, dejando sólo tres elementos. Por último, el programa comprueba el `23` (que coincide con la clave de búsqueda) y devuelve el índice `1`.

Eficiencia de la búsqueda binaria

En el peor de los casos, el proceso de buscar en un arreglo ordenado de 1023 elementos sólo requiere 10 comparaciones cuando se utiliza una búsqueda binaria. Al dividir 1023 entre 2 en forma repetida (ya que después de cada comparación podemos eliminar la mitad del arreglo) y redondear (porque también eliminamos el elemento medio), se producen los valores 511, 255, 127, 63, 31, 15, 7, 3, 1 y 0. El número 1023 ($2^{10} - 1$) se divide entre 2 sólo 10 veces para obtener el valor 0, que indica que no hay más elementos para probar. La división entre 2

```
1  // Fig. 16.5: PruebaBusquedaBinaria.java
2  // Usa la búsqueda binaria para localizar un elemento en un arreglo.
3  import java.util.Scanner;
4
5  public class PruebaBusquedaBinaria
6  {
7     public static void main( String args[] )
8     {
9        // crea un objeto Scanner para recibir datos de entrada
10       Scanner entrada = new Scanner( System.in );
11
12       int enteroABuscar; // clave de búsqueda
13       int posicion; // ubicación de la clave de búsqueda en el arreglo
14
15       // crea un arreglo y lo muestra en pantalla
16       ArregloBinario arregloBusqueda = new ArregloBinario( 15 );
17       System.out.println( arregloBusqueda );
18
19       // obtiene la entrada del usuario
20       System.out.print(
21          "Escriba un valor entero (-1 para salir): ");
22       enteroABuscar = entrada.nextInt(); // lee un entero del usuario
23       System.out.println();
24
25       // recibe un entero como entrada en forma repetida; -1 termina el programa
26       while ( enteroABuscar != -1 )
```

Figura 16.5 | La clase `PruebaBusquedaBinaria`. (Parte 1 de 2).

```
        {
           // usa la búsqueda binaria para tratar de encontrar el entero
           posicion = arregloBusqueda.busquedaBinaria( enteroABuscar );

           // el valor de retorno -1 indica que no se encontró el entero
           if ( posicion == -1 )
              System.out.println( "El entero " + enteroABuscar +
                 " no se encontro.\n" );
           else
              System.out.println( "El entero " + enteroABuscar +
                 " se encontro en la posicion " + posicion + ".\n" );

           // obtiene entrada del usuario
           System.out.print(
              "Escriba un valor entero (-1 para salir): " );
           enteroABuscar = entrada.nextInt(); // lee un entero del usuario
           System.out.println();
        } // fin de while
     } // fin de main
  } // fin de la clase PruebaBusquedaBinaria
```

```
13 23 24 34 35 36 38 42 47 51 68 74 75 85 97

Escriba un valor entero (-1 para salir): 23

13 23 24 34 35 36 38 42 47 51 68 74 75 85 97
                      *
13 23 24 34 35 36 38
          *
13 23 24
    *
El entero 23 se encontro en la posicion 1.

Escriba un valor entero (-1 para salir): 75

13 23 24 34 35 36 38 42 47 51 68 74 75 85 97
                      *
                        47 51 68 74 75 85 97
                                  *
                                    75 85 97
                                        *
                                    75
                                     *
El entero 75 se encontro en la posicion 12.

Escriba un valor entero (-1 para salir): 52

13 23 24 34 35 36 38 42 47 51 68 74 75 85 97
                      *
                        47 51 68 74 75 85 97
                                  *
                        47 51 68
                            *
                              68
                               *
El entero 52 no se encontro.

Escriba un valor entero (-1 para salir): -1
```

Figura 16.5 | La clase `PruebaBusquedaBinaria`. (Parte 2 de 2).

equivale a una comparación en el algoritmo de búsqueda binaria. Por ende, un arreglo de 1,048,575 ($2^{20} - 1$) elementos requiere un máximo de 20 comparaciones para encontrar la clave, y un arreglo de más de mil millones de elementos requiere un máximo de 30 comparaciones para encontrar la clave. Ésta es una enorme mejora en el rendimiento, en comparación con la búsqueda lineal. Para un arreglo de mil millones de elementos, ésta es una diferencia entre un promedio de 500 millones de comparaciones para la búsqueda lineal, ¡y un máximo de sólo 30 comparaciones para la búsqueda binaria! El número máximo de comparaciones necesarias para la búsqueda binaria de cualquier arreglo ordenado es el exponente de la primera potencia de 2 mayor que el número de elementos en el arreglo, que se representa como $\log_2 n$. Todos los logaritmos crecen aproximadamente a la misma proporción, por lo que en notación Big O se puede omitir la base. Esto produce un valor Big O de ***O*(log *n*)** para una búsqueda binaria, que también se conoce como **tiempo de ejecución logarítmico**.

16.3 Algoritmos de ordenamiento

El ordenamiento de datos (es decir, colocar los datos en cierto orden específico, como ascendente o descendente) es una de las aplicaciones computacionales más importantes. Un banco ordena todos los cheques por número de cuenta, de manera que pueda preparar instrucciones bancarias individuales al final de cada mes. Las compañías telefónicas ordenan sus listas de cuentas por apellido paterno y luego por primer nombre, para facilitar el proceso de buscar números telefónicos. Casi cualquier organización debe ordenar datos, y a menudo cantidades masivas de ellos. El ordenamiento de datos es un problema intrigante, que requiere un uso intensivo de la computadora, y ha atraído un enorme esfuerzo de investigación.

Un punto importante a comprender acerca del ordenamiento es que el resultado final (los datos ordenados) será el mismo, sin importar qué algoritmo se utilice para ordenar los datos. La elección del algoritmo sólo afecta al tiempo de ejecución y el uso que haga el programa de la memoria. En el resto del capítulo se introducen tres algoritmos de ordenamiento comunes. Los primeros dos (ordenamiento por selección y ordenamiento por inserción) son simples de programar, pero ineficientes. El último algoritmo (ordenamiento por combinación) es más rápido que el ordenamiento por selección y el ordenamiento por inserción, pero más difícil de programar. Nos enfocaremos en ordenar arreglos de datos de tipos primitivos, principalmente valores `int`. Es posible ordenar arreglos de objetos de clases también. En la sección 19.6.1 hablaremos sobre esto.

16.3.1 Ordenamiento por selección

El **ordenamiento por selección** es un algoritmo de ordenamiento simple, pero ineficiente. En la primera iteración del algoritmo se selecciona el elemento más pequeño en el arreglo, y se intercambia con el primer elemento. En la segunda iteración se selecciona el segundo elemento más pequeño (que viene siendo el elemento más pequeño de los elementos restantes) y se intercambia con el segundo elemento. El algoritmo continúa hasta que en la última iteración se selecciona el segundo elemento más grande y se intercambia con el índice del segundo al último, dejando el elemento más grande en el último índice. Después de la *i*-ésima iteración, los *i* elementos más pequeños del arreglo se ordenarán en forma ascendente, en los primeros *i* elementos del arreglo.

Como ejemplo, considere el siguiente arreglo:

```
34   56   4   10   77   51   93   30   5   52
```

Un programa que implemente el ordenamiento por selección primero determinará el elemento más pequeño (4) de este arreglo, que está contenido en el índice 2. El programa intercambia 4 con 34, dando el siguiente resultado:

```
4   56   34   10   77   51   93   30   5   52
```

Después el programa determina el valor más pequeño del resto de los elementos (todos los elementos excepto el 4), que es 5 y está contenido en el índice 8. El programa intercambia el 5 con el 56, dando el siguiente resultado:

```
4   5   34   10   77   51   93   30   56   52
```

En la tercera iteración, el programa determina el siguiente valor más pequeño (10) y lo intercambia con el 34.

```
4   5   10   34   77   51   93   30   56   52
```

El proceso continúa hasta que el arreglo está completamente ordenado.

```
4   5   10   30   34   51   52   56   77   93
```

Observe que después de la primera iteración, el elemento más pequeño está en la primera posición. Después de la segunda iteración los dos elementos más pequeños están en orden, en las primeras dos posiciones. Después de la tercera iteración los tres elementos más pequeños están en orden, en las primeras tres posiciones.

En la figura 16.6 se declara la clase `OrdenamientoSeleccion`. Esta clase tiene dos variables de instancia `private`: un arreglo de valores `int` llamado `datos`, y un objeto `static Random` para generar enteros aleatorios y llenar el arreglo. Cuando se crea una instancia de un objeto de la clase `OrdenamientoSeleccion`, el constructor (líneas 12 a 19) crea e inicializa el arreglo `datos` con valores `int` aleatorios, en el rango de `10 a 99`.

```
1  // Fig. 16.6: OrdenamientoSeleccion.java
2  // Clase que crea un arreglo lleno con enteros aleatorios.
3  // Proporciona un método para ordenar el arreglo mediante el ordenamiento por selección.
4  import java.util.Random;
5
6  public class OrdenamientoSeleccion
7  {
8     private int[] datos; // arreglo de valores
9     private static Random generador = new Random();
10
11    // crea un arreglo de un tamaño dado y lo llena con enteros aleatorios
12    public OrdenamientoSeleccion( int tamanio )
13    {
14       datos = new int[ tamanio ]; // crea espacio para el arreglo
15
16       // llena el arreglo con enteros aleatorios en el rango de 10 a 99
17       for ( int i = 0; i < tamanio; i++ )
18          datos[ i ] = 10 + generador.nextInt( 90 );
19    } // fin del constructor de OrdenamientoSeleccion
20
21    // ordena el arreglo usando el ordenamiento por selección
22    public void ordenar()
23    {
24       int masPequenio; // índice del elemento más pequeño
25
26       // itera a través de datos.length - 1 elementos
27       for ( int i = 0; i < datos.length - 1; i++ )
28       {
29          masPequenio = i; // primer índice del resto del arreglo
30
31          // itera para buscar el índice del elemento más pequeño
32          for ( int indice = i + 1; indice < datos.length; indice++ )
33             if ( datos[ indice ] < datos[ masPequenio ] )
34                masPequenio = indice;
35
36          intercambiar( i, masPequenio ); // intercambia el elemento más pequeño en la
            posición
37          imprimirPasada( i + 1, masPequenio ); // imprime la pasada del algoritmo
38       } // fin de for exterior
39    } // fin del método ordenar
40
41    // método ayudante para intercambiar los valores de dos elementos
42    public void intercambiar( int primero, int segundo )
43    {
44       int temporal = datos[ primero ]; // almacena primero en temporal
```

Figura 16.6 | La clase `OrdenamientoSeleccion`. (Parte 1 de 2).

```
45          datos[ primero ] = datos[ segundo ]; // sustituye primero con segundo
46          datos[ segundo ] = temporal; // coloca temporal en segundo
47       } // fin del método intercambiar
48
49       // imprime una pasada del algoritmo
50       public void imprimirPasada( int pasada, int indice )
51       {
52          System.out.print( String.format( "despues de pasada %2d: ", pasada ) );
53
54          // imprime elementos hasta el elemento seleccionado
55          for ( int i = 0; i < indice; i++ )
56             System.out.print( datos[ i ] + "  " );
57
58          System.out.print( datos[ indice ] + "* " ); // indica intercambio
59
60          // termina de imprimir el arreglo en pantalla
61          for ( int i = indice + 1; i < datos.length; i++ )
62             System.out.print( datos[ i ] + "  " );
63
64          System.out.print( "\n                " ); // para alineación
65
66          // indica la cantidad del arreglo que está almacenada
67          for( int j = 0; j < pasada; j++ )
68             System.out.print( "--  " );
69          System.out.println( "\n" ); // agrega fin de línea
70       } // fin del método imprimirPasada
71
72       // método para imprimir los valores del arreglo
73       public String toString()
74       {
75          StringBuilder temporal = new StringBuilder();
76
77          // itera a través del arreglo
78          for ( int elemento : datos )
79             temporal.append( elemento + "  " );
80
81          temporal.append( "\n" ); // agrega carácter de nueva línea
82          return temporal.toString();
83       } // fin del método toString
84    } // fin de la clase OrdenamientoSeleccion
```

Figura 16.6 | La clase `OrdenamientoSeleccion`. (Parte 2 de 2).

En las líneas 22 a 39 se declara el método `ordenar`. En la línea 24 se declara la variable `masPequenio`, que almacenará el índice del elemento más pequeño en el resto del arreglo. En las líneas 27 a 38 se itera `datos.length – 1` veces. En la línea 29 se inicializa el índice del elemento más pequeño con el elemento actual. En las líneas 32 a 34 se itera a través del resto de los elementos en el arreglo. Para cada uno de estos elementos, en la línea 33 se compara su valor con el valor del elemento más pequeño. Si el elemento actual es menor que el elemento más pequeño, en la línea 34 se asigna el índice del elemento actual a `masPequenio`. Cuando termine este ciclo, `masPequenio` contendrá el índice del elemento más pequeño en el resto del arreglo. En la línea 36 se hace una llamada al método `intercambiar` (líneas 42 a 47) para colocar el elemento restante más pequeño en la siguiente posición en el arreglo.

En la línea 9 de la figura 16.7 se crea un objeto `OrdenamientoSeleccion` con `10` elementos. En la línea 12 se hace una llamada implícita al método `toString` para imprimir el objeto desordenado en pantalla. En la línea 14 se hace una llamada al método `ordenar` (líneas 22 a 39 de la figura 16.6), el cual ordena los elementos mediante el ordenamiento por selección. Después, en las líneas 16 y 17 se imprime el objeto ordenado en pantalla. En la salida de este programa se utilizan guiones cortos para indicar la porción del arreglo que se ordenó después de

cada pasada. Se coloca un asterisco enseguida de la posición del elemento que se intercambió con el elemento más pequeño en esa pasada. En cada pasada, el elemento enseguida del asterisco y el elemento por encima del conjunto de guiones cortos de más a la derecha fueron los dos valor es que se intercambiaron.

```java
// Fig. 16.7: PruebaOrdenamientoSeleccion.java
// Prueba la clase de ordenamiento por selección.

public class PruebaOrdenamientoSeleccion
{
   public static void main( String[] args )
   {
      // crea objeto para realizar el ordenamiento por selección
      OrdenamientoSeleccion arregloOrden = new OrdenamientoSeleccion( 10 );

      System.out.println( "Arreglo desordenado:" );
      System.out.println( arregloOrden ); // imprime arreglo desordenado

      arregloOrden.ordenar(); // ordena el arreglo

      System.out.println( "Arreglo ordenado:");
      System.out.println( arregloOrden ); // imprime el arreglo ordenado
   } // fin de main
} // fin de la clase PruebaOrdenamientoSeleccion
```

```
Arreglo desordenado:
45  47  61  92  74  12  21  30  72  33

despues de pasada  1: 12  47  61  92  74  45* 21  30  72  33
                      --

despues de pasada  2: 12  21  61  92  74  45  47* 30  72  33
                      --  --

despues de pasada  3: 12  21  30  92  74  45  47  61* 72  33
                      --  --  --

despues de pasada  4: 12  21  30  33  74  45  47  61  72  92*
                      --  --  --  --

despues de pasada  5: 12  21  30  33  45  74* 47  61  72  92
                      --  --  --  --  --

despues de pasada  6: 12  21  30  33  45  47  74* 61  72  92
                      --  --  --  --  --  --

despues de pasada  7: 12  21  30  33  45  47  61  74* 72  92
                      --  --  --  --  --  --  --

despues de pasada  8: 12  21  30  33  45  47  61  72  74* 92
                      --  --  --  --  --  --  --  --

despues de pasada  9: 12  21  30  33  45  47  61  72  74 92*
                      --  --  --  --  --  --  --  --  --

Arreglo ordenado:
12  21  30  33  45  47  61  72  74  92
```

Figura 16.7 | La clase `PruebaOrdenamientoSeleccion`.

Eficiencia del ordenamiento por selección

El algoritmo de ordenamiento por selección se ejecuta en un tiempo igual a $O(n^2)$. El método `ordenar` en las líneas 22 a 39 de la figura 16.6, que implementa el algoritmo de ordenamiento por selección, contiene dos ciclos `for`. El ciclo `for` exterior (líneas 27 a 38) itera a través de los primeros $n - 1$ elementos en el arreglo, intercambiando el elemento más pequeño restante a su posición ordenada. El ciclo `for` interior (líneas 32 a 34) itera a través de cada elemento en el arreglo restante, buscando el elemento más pequeño. Este ciclo se ejecuta $n - 1$ veces durante la primera iteración del ciclo exterior, $n - 2$ veces durante la segunda iteración, después $n - 3, \ldots, 3, 2, 1$. Este ciclo interior iterará un total de $n(n - 1)/2$ o $(n^2 - n)/2$. En notación Big O, los términos más pequeños se eliminan y las constantes se ignoran, lo cual nos deja un valor Big O final de $O(n^2)$.

16.3.2 Ordenamiento por inserción

El **ordenamiento por inserción** es otro algoritmo de ordenamiento simple, pero ineficiente. En la primera iteración de este algoritmo se toma el segundo elemento en el arreglo y, si es menor que el primero, se intercambian. En la segunda iteración se analiza el tercer elemento y se inserta en la posición correcta, con respecto a los primeros dos elementos, de manera que los tres elementos estén ordenados. En la *i*-ésima iteración de este algoritmo, los primeros *i* elementos en el arreglo original estarán ordenados.

Considere como ejemplo el siguiente arreglo. [*Nota:* este arreglo es idéntico al que se utiliza en las discusiones sobre el ordenamiento por selección y el ordenamiento por combinación].

```
34   56   4   10   77   51   93   30   5   52
```

Un programa que implemente el algoritmo de ordenamiento por inserción primero analizará los primeros dos elementos del arreglo, 34 y 56. Estos dos elementos ya se encuentran ordenados, por lo que el programa continúa (si estuvieran desordenados, el programa los intercambiaría).

En la siguiente iteración, el programa analiza el tercer valor, 4. Este valor es menor que 56, por lo que el programa almacena el 4 en una variable temporal y mueve el 56 un elemento a la derecha. Después, el programa comprueba y determina que 4 es menor que 34, por lo que mueve el 34 un elemento a la derecha. Ahora el programa ha llegado al principio del arreglo, por lo que coloca el 4 en el elemento cero. Entonces, el arreglo es ahora

```
4   34   56   10   77   51   93   30   5   52
```

En la siguiente iteración, el programa almacena el valor 10 en una variable temporal. Después el programa compara el 10 con el 56, y mueve el 56 un elemento a la derecha, ya que es mayor que 10. Luego, el programa compara 10 y 34, y mueve el 34 un elemento a la derecha. Cuando el programa compara el 10 con el 4, observa que el primero es mayor que el segundo, por lo cual coloca el 10 en el elemento 1. Ahora el arreglo es

```
4   10   34   56   77   51   93   30   5   52
```

Utilizando este algoritmo, en la *i*-ésima iteración, los primeros *i* elementos del arreglo original están ordenados. Tal vez no se encuentren en sus posiciones finales, debido a que puede haber valores más pequeños en posiciones más adelante en el arreglo.

En la figura 16.8 se declara la clase `OrdenamientoInsercion`. En las líneas 22 a 46 se declara el método `ordenar`. En la línea 24 se declara la variable `insercion`, la cual contiene el elemento que insertaremos mientras movemos los demás elementos. En las líneas 27 a 45 se itera a través de `datos.length - 1` elementos en el arreglo.

En cada iteración, en la línea 30 se almacena en `insercion` el valor del elemento que se insertará en la parte ordenada del arreglo. En la línea 33 se declara e inicializa la variable `moverElemento`, que lleva la cuenta de la posición en la que se insertará el elemento. En las líneas 36 a 41 se itera para localizar la posición correcta en la que debe insertarse el elemento. El ciclo terminará, ya sea cuando el programa llegue a la parte frontal del arreglo, o cuando llegue a un elemento que sea menor que el valor a insertar. En la línea 39 se mueve un elemento a la derecha, y en la línea 40 se decrementa la posición en la que se insertará el siguiente elemento. Una vez que termina el ciclo, en la línea 43 se inserta el elemento en su posición. La figura 16.9 es igual que la figura 16.7, sólo que crea y utiliza un objeto `OrdenamientoInsercion`. En la salida de este programa se utilizan guiones cortos para indicar la parte del arreglo que se ordena después de cada pasada. Se coloca un asterisco enseguida del elemento que se insertó en su posición en esa pasada.

```
// Fig. 16.8: OrdenamientoInsercion.java
// Clase que crea un arreglo lleno de enteros aleatorios.
// Proporciona un método para ordenar el arreglo mediante el ordenamiento por
inserción.
import java.util.Random;

public class OrdenamientoInsercion
{
   private int[] datos; // arreglo de valores
   private static Random generador = new Random();

   // crea un arreglo de un tamaño dado y lo llena con enteros aleatorios
   public OrdenamientoInsercion( int tamanio )
   {
      datos = new int[ tamanio ]; // crea espacio para el arreglo

      // llena el arreglo con enteros aleatorios en el rango de 10 a 99
      for ( int i = 0; i < tamanio; i++ )
         datos[ i ] = 10 + generador.nextInt( 90 );
   } // fin del constructor de OrdenamientoInsercion

   // ordena el arreglo usando el ordenamiento por inserción
   public void sort()
   {
      int insercion; // variable temporal para contener el elemento a insertar

      // itera a través de datos.length - 1 elementos
      for ( int siguiente = 1; siguiente < datos.length; siguiente++ )
      {
         // almacena el valor en el elemento actual
         insercion = datos[ siguiente ];

         // inicializa ubicación para colocar el elemento
         int moverElemento = siguiente;

         // busca un lugar para colocar el elemento actual
         while ( moverElemento > 0 && datos[ moverElemento - 1 ] > insercion )
         {
           // desplaza el elemento una posición a la derecha
           datos[ moverElemento ] = datos[ moverElemento - 1 ];
           moverElemento--;
         } // fin de while

         datos[ moverElemento ] = insercion; // coloca el elemento insertado
         imprimirPasada( siguiente, moverElemento ); // imprime la pasada del algoritmo
      } // fin de for
   } // fin del método ordenar

   // imprime una pasada del algoritmo
   public void imprimirPasada( int pasada, int indice )
   {
      System.out.print( String.format( "despues de pasada %2d: ", pasada ) );

      // imprime los elementos hasta el elemento intercambiado
      for ( int i = 0; i < indice; i++ )
         System.out.print( datos[ i ] + "  " );

      System.out.print( datos[ indice ] + "* " ); // indica intercambio

```

Figura 16.8 | La clase `OrdenamientoInsercion`. (Parte 1 de 2).

```java
      // termina de imprimir el arreglo en pantalla
      for ( int i = indice + 1; i < datos.length; i++ )
         System.out.print( datos[ i ] + "     " );

      System.out.print( "\n            " ); // para alineación

      // indica la cantidad del arreglo que está ordenado
      for( int i = 0; i <= pasada; i++ )
         System.out.print( "--  " );
      System.out.println( "\n" ); // agrega fin de línea
   } // fin del método imprimirPasada

   // método para mostrar los valores del arreglo en pantalla
   public String toString()
   {
      StringBuilder temporal = new StringBuilder();

      // itera a través del arreglo
      for ( int elemento : datos )
         temporal.append( elemento + "  " );

      temporal.append( "\n" ); // agrega carácter de fin de línea
      return temporal.toString();
   } // fin del método toString
} // fin de la clase OrdenamientoInsercion
```

Figura 16.8 | La clase `OrdenamientoInsercion`. (Parte 2 de 2).

```java
// Fig. 16.9: PruebaOrdenamientoInsercion.java
// Prueba la clase de ordenamiento por inserción.

public class PruebaOrdenamientoInsercion
{
   public static void main( String[] args )
   {
      // crea objeto para realizar el ordenamiento por inserción
      OrdenamientoInsercion arregloOrden = new OrdenamientoInsercion( 10 );

      System.out.println( "Arreglo desordenado:" );
      System.out.println( arregloOrden ); // imprime el arreglo desordenado

      arregloOrden.sort(); // ordena el arreglo

      System.out.println( "Arreglo ordenado:" );
      System.out.println( arregloOrden ); // imprime el arreglo ordenado
   } // fin de main
} // fin de la clase PruebaOrdenamientoInsercion
```

```
Arreglo desordenado:
19  42  68  88  76  54  16  99  54  26

despues de pasada  1: 19  42* 68  88  76  54  16  99  54  26
                      --  --

despues de pasada  2: 19  42  68* 88  76  54  16  99  54  26
                      --  --  --
despues de pasada  3: 19  42  68  88* 76  54  16  99  54  26
                      --  --  --  --
```

Figura 16.9 | La clase `PruebaOrdenamientoInsercion`. (Parte 1 de 2).

```
despues de pasada  4: 19  42  68  76* 88  54  16  99  54  26
                      --  --  --  --  --

despues de pasada  5: 19  42  54* 68  76  88  16  99  54  26
                      --  --  --  --  --  --

despues de pasada  6: 16* 19  42  54  68  76  88  99  54  26
                      --  --  --  --  --  --  --

despues de pasada  7: 16  19  42  54  68  76  88  99* 54  26
                      --  --  --  --  --  --  --  --

despues de pasada  8: 16  19  42  54  54* 68  76  88  99  26
                      --  --  --  --  --  --  --  --  --

despues de pasada  9: 16  19  26* 42  54  54  68  76  88  99
                      --  --  --  --  --  --  --  --  --  --

Arreglo ordenado:
16  19  26  42  54  54  68  76  88  99
```

Figura 16.9 | La clase `PruebaOrdenamientoInsercion`. (Parte 2 de 2).

Eficiencia del ordenamiento por inserción

El algoritmo de ordenamiento por inserción también se ejecuta en un tiempo igual a $O(n^2)$. Al igual que el ordenamiento por selección, la implementación del ordenamiento por inserción (líneas 22 a 46 de la figura 16.8) contiene dos ciclos. El ciclo `for` (líneas 27 a 45) itera `datos.length - 1` veces, insertando un elemento en la posición apropiada en los elementos ordenados hasta ahora. Para los fines de esta aplicación, `datos.length - 1` es equivalente a $n - 1$ (ya que `datos.length` es el tamaño del arreglo). El ciclo `while` (líneas 36 a 41) itera a través de los anteriores elementos en el arreglo. En el peor de los casos, el ciclo `while` requerirá $n - 1$ comparaciones. Cada ciclo individual se ejecuta en un tiempo $O(n)$. En notación Big O, los ciclos anidados indican que debemos multiplicar el número de comparaciones. Para cada iteración de un ciclo exterior, habrá cierto número de iteraciones en el ciclo interior. En este algoritmo, para cada $O(n)$ iteraciones del ciclo exterior, habrá $O(n)$ iteraciones del ciclo interior. Al multiplicar estos valores se produce un valor Big O de $O(n^2)$.

16.3.3 Ordenamiento por combinación

El **ordenamiento por combinación** es un algoritmo de ordenamiento eficiente, pero en concepto es más complejo que los ordenamientos de selección y de inserción. Para ordenar un arreglo, el algoritmo de ordenamiento por combinación lo divide en dos subarreglos de igual tamaño, ordena cada subarreglo y después los combina en un arreglo más grande. Con un número impar de elementos, el algoritmo crea los dos subarreglos de tal forma que uno tenga más elementos que el otro.

La implementación del ordenamiento por combinación en este ejemplo es recursiva. El caso base es un arreglo con un elemento que, desde luego, está ordenado, por lo que el ordenamiento por combinación regresa de inmediato en este caso. El paso recursivo divide el arreglo en dos piezas de un tamaño aproximadamente igual, las ordena en forma recursiva y después combina los dos arreglos ordenados en un arreglo ordenado de mayor tamaño.

Suponga que el algoritmo ya ha combinado arreglos más pequeños para crear los arreglos ordenados A:

```
4    10    34    56    77
```

y B:

```
5    30    51    52    93
```

El ordenamiento por combinación combina estos dos arreglos en un arreglo ordenado de mayor tamaño. El elemento más pequeño en A es 4 (que se encuentra en el índice cero de A). El elemento más pequeño en B es 5 (que

se encuentra en el índice cero de B). Para poder determinar el elemento más pequeño en el arreglo más grande, el algoritmo compara 4 y 5. El valor de A es más pequeño, por lo que el 4 se convierte en el primer elemento del arreglo combinado. El algoritmo continúa, para lo cual compara 10 (el segundo elemento en A) con 5 (el primer elemento en B). El valor de B es más pequeño, por lo que 5 se convierte en el segundo elemento del arreglo más grande. El algoritmo continúa comparando 10 con 30, en donde 10 se convierte en el tercer elemento del arreglo, y así en lo sucesivo.

En las líneas 22 a 25 de la figura 16.10 se declara el método `ordenar`. En la línea 24 se hace una llamada al método `ordenarArreglo` con `0` y `datos.length – 1` como los argumentos (que corresponden a los índices inicial y final, respectivamente, del arreglo que se ordenará). Estos valores indican al método `ordenarArreglo` que debe operar en todo el arreglo completo.

```
1  // Fig. 16.10: OrdenamientoCombinacion.java
2  // Clase que crea un arreglo lleno con enteros aleatorios.
3  // Proporciona un método para ordenar el arreglo mediante el ordenamiento por combinación.
4  import java.util.Random;
5
6  public class OrdenamientoCombinacion
7  {
8     private int[] datos; // arreglo de valores
9     private static Random generador = new Random();
10
11    // crea un arreglo de un tamaño dado y lo llena con enteros aleatorios
12    public OrdenamientoCombinacion( int tamanio )
13    {
14       datos = new int[ tamanio ]; // crea espacio para el arreglo
15
16       // llena el arreglo con enteros aleatorios en el rango de 10 a 99
17       for ( int i = 0; i < tamanio; i++ )
18          datos[ i ] = 10 + generador.nextInt( 90 );
19    } // fin del constructor de OrdenamientoCombinacion
20
21    // llama al método de división recursiva para comenzar el ordenamiento por combinación
22    public void ordenar()
23    {
24       ordenarArreglo( 0, datos.length - 1 ); // divide todo el arreglo
25    } // fin del método ordenar
26
27    // divide el arreglo, ordena los subarreglos y los combina en un arreglo ordenado
28    private void ordenarArreglo( int inferior, int superior )
29    {
30       // evalúa el caso base; el tamaño del arreglo es igual a 1
31       if ( ( superior - inferior ) >= 1 ) // si no es el caso base
32       {
33          int medio1 = ( inferior + superior ) / 2; // calcula el elemento medio del arreglo
34          int medio2 = medio1 + 1; // calcula el siguiente elemento arriba
35
36          // imprime en pantalla el paso de división
37          System.out.println( "division:   " + subarreglo( inferior, superior ) );
38          System.out.println( "            " + subarreglo( inferior, medio1 ) );
39          System.out.println( "            " + subarreglo( medio2, superior ) );
40          System.out.println();
41
42          // divide el arreglo a la mitad; ordena cada mitad (llamadas recursivas)
43          ordenarArreglo( inferior, medio1 ); // primera mitad del arreglo
44          ordenarArreglo( medio2, superior ); // segunda mitad del arreglo
45
```

Figura 16.10 | La clase `OrdenamientoCombinacion`. (Parte 1 de 3).

```
         // combina dos arreglos ordenados después de que regresan las llamadas de división
         combinar ( inferior, medio1, medio2, superior );
      } // fin de if
   } // fin del método ordenarArreglo

   // combina dos subarreglos ordenados en un subarreglo ordenado
   private void combinar( int izquierdo, int medio1, int medio2, int derecho )
   {
      int indiceIzq = izquierdo; // índice en subarreglo izquierdo
      int indiceDer = medio2; // índice en subarreglo derecho
      int indiceCombinado = izquierdo; // índice en arreglo de trabajo temporal
      int[] combinado = new int[ datos.length ]; // arreglo de trabajo

      // imprime en pantalla los dos subarreglos antes de combinarlos
      System.out.println( "combinacion: " + subarreglo( izquierdo, medio1 ) );
      System.out.println( "             " + subarreglo( medio2, derecho ) );

      // combina los arreglos hasta llegar al final de uno de ellos
      while ( indiceIzq <= medio1 && indiceDer <= derecho )
      {
         // coloca el menor de dos elementos actuales en el resultado
         // y lo mueve al siguiente espacio en los arreglos
         if ( datos[ indiceIzq ] <= datos[ indiceDer ] )
            combinado[ indiceCombinado++ ] = datos[ indiceIzq++ ];
         else
            combinado[ indiceCombinado++ ] = datos[ indiceDer++ ];
      } // fin de while

      // si el arreglo izquierdo está vacío
      if ( indiceIzq == medio2 )
         // copia el resto del arreglo derecho
         while ( indiceDer <= derecho )
            combinado[ indiceCombinado++ ] = datos[ indiceDer++ ];
      else // el arreglo derecho está vacío
         // copia el resto del arreglo izquierdo
         while ( indiceIzq <= medio1 )
            combinado[ indiceCombinado++ ] = datos[ indiceIzq++ ];

      // copia los valores de vuelta al arreglo original
      for ( int i = izquierdo; i <= derecho; i++ )
         datos[ i ] = combinado[ i ];

      // imprime en pantalla el arreglo combinado
      System.out.println( "          " + subarreglo( izquierdo, derecho ) );
      System.out.println();
   } // fin del método combinar

   // método para imprimir en pantalla ciertos valores en el arreglo
   public String subarreglo( int inferior, int superior )
   {
      StringBuilder temporal = new StringBuilder();

      // imprime en pantalla espacios para la alineación
      for ( int i = 0; i < inferior; i++ )
         temporal.append( "   " );

      // imprime en pantalla el resto de los elementos en el arreglo
      for ( int i = inferior; i <= superior; i++ )
```

Figura 16.10 | La clase `OrdenamientoCombinacion`. (Parte 2 de 3).

```
104            temporal.append( " " + datos[ i ] );
105
106         return temporal.toString();
107      } // fin del método subarreglo
108
109      // método para imprimir los valores en el arreglo
110      public String toString()
111      {
112         return subarreglo( 0, datos.length - 1 );
113      } // fin del método toString
114   } // fin de la clase OrdenamientoCombinacion
```

Figura 16.10 | La clase `OrdenamientoCombinacion`. (Parte 3 de 3).

El método `ordenarArreglo` se declara en las líneas 28 a 49. En la línea 31 se evalúa el caso base. Si el tamaño del arreglo es `1`, ya está ordenado, por lo que el método regresa de inmediato. Si el tamaño del arreglo es mayor que `1`, el método divide el arreglo en dos, llama en forma recursiva al método `ordenarArreglo` para ordenar los dos subarreglos y después los combina. En la línea 43 se hace una llamada recursiva al método `ordenarArreglo` en la primera mitad del arreglo, y en la línea 44 se hace una llamada recursiva al método `ordenarArreglo` en la segunda mitad del arreglo. Cuando regresan estas dos llamadas al método, cada mitad del arreglo se ha ordenado. En la línea 47 se hace una llamada al método `combinar` (líneas 52 a 91) con las dos mitades del arreglo, para combinar los dos arreglos ordenados en un arreglo ordenado más grande.

En las líneas 64 a 72 en el método `combinar` se itera hasta que el programa llega al final de cualquiera de los subarreglos. En la línea 68 se evalúa cuál elemento al principio de los arreglos es más pequeño. Si el elemento en el arreglo izquierdo es más pequeño, en la línea 69 se coloca el elemento en su posición en el arreglo combinado. Si el elemento en el arreglo derecho es más pequeño, en la línea 71 se coloca en su posición en el arreglo combinado. Cuando el ciclo `while` ha terminado (línea 72), un subarreglo completo se coloca en el arreglo combinado, pero el otro subarreglo aún contiene datos. En la línea 75 se evalúa si el arreglo izquierdo ha llegado al final. De ser así, en las líneas 77 y 78 se llena el arreglo combinado con los elementos del arreglo derecho. Si el arreglo izquierdo no ha llegado al final, entonces el arreglo derecho debe haber llegado, por lo que en las líneas 81 y 82 se llena el arreglo combinado con los elementos del arreglo izquierdo. Por último, en las líneas 85 y 86 se copia el arreglo combinado en el arreglo original. En la figura 16.11 se crea y se utiliza un objeto `OrdenamientoCombinado`. Los fascinantes resultados de este programa muestran las divisiones y combinaciones que realiza el ordenamiento por combinación, mostrando también el progreso del ordenamiento en cada paso del algoritmo. Bien vale la pena el tiempo que usted invierta al recorrer estos resultados paso a paso, para comprender por completo este elegante algoritmo de ordenamiento.

```
// Fig. 16.11: PruebaOrdenamientoCombinacion.java
// Prueba la clase de ordenamiento por combinación.

public class PruebaOrdenamientoCombinacion
{
   public static void main( String[] args )
   {
      // crea un objeto para realizar el ordenamiento por combinación
      OrdenamientoCombinacion arregloOrden = new OrdenamientoCombinacion( 10 );

      // imprime el arreglo desordenado
      System.out.println( "Desordenado:" + arregloOrden + "\n" );

      arregloOrden.ordenar(); // ordena el arreglo

      // imprime el arreglo ordenado
```

Figura 16.11 | La clase `PruebaOrdenamientoCombinacion`. (Parte 1 de 3).

```
      System.out.println( "Ordenado:    " + arregloOrden );
   } // fin de main
} // fin de la clase PruebaOrdenamientoCombinacion
```

```
Desordenado: 67 43 32 76 19 75 78 73 57 94

division:    67 43 32 76 19 75 78 73 57 94
             67 43 32 76 19
                            75 78 73 57 94

Desordenado: 95 30 48 23 68 78 19 23 14 33

division:    95 30 48 23 68 78 19 23 14 33
             95 30 48 23 68
                            78 19 23 14 33

division:    95 30 48 23 68
             95 30 48
                      23 68

division:    95 30 48
             95 30
                   48

division:    95 30
             95
                30

combinacion: 95
                30
             30 95

combinacion: 30 95
                   48
             30 48 95

division:             23 68
                      23
                         68

combinacion:          23
                         68
                      23 68

combinacion: 30 48 95
                      23 68
             23 30 48 68 95
division:                   78 19 23 14 33
                            78 19 23
                                     14 33

division:                   78 19 23
                            78 19
                                  23

division:                   78 19
                            78
                               19

combinacion:                78
                               19
                            19 78
```

Figura 16.11 | La clase `PruebaOrdenamientoCombinacion`. (Parte 2 de 3).

```
combinacion:            19 78
                              23
                        19 23 78

division:                        14 33
                                 14
                                    33
combinacion:                     14
                                    33
                                 14 33

combinacion:            19 23 78
                                 14 33
                        14 19 23 33 78

combinacion: 23 30 48 68 95
                        14 19 23 33 78
             14 19 23 23 30 33 48 68 78 95

Ordenado:    14 19 23 23 30 33 48 68 78 95
```

Figura 16.11 | La clase `PruebaOrdenamientoCombinacion`. (Parte 3 de 3).

Eficiencia del ordenamiento por combinación

El ordenamiento por combinación es un algoritmo mucho más eficiente que el de inserción o el de selección. Considere la primera llamada (no recursiva) al método `ordenarArreglo`. Esto produce dos llamadas recursivas al método `ordenarArreglo` con subarreglos, cada uno de los cuales tiene un tamaño aproximado de la mitad del arreglo original, y una sola llamada al método `combinar`. Esta llamada a `combinar` requiere, a lo más, $n - 1$ comparaciones para llenar el arreglo original, que es $O(n)$. (Recuerde que se puede elegir cada elemento en el arreglo mediante la comparación de un elemento de cada uno de los subarreglos). Las dos llamadas al método `ordenarArreglo` producen cuatro llamadas recursivas más al método `ordenarArreglo`, cada una con un subarreglo de un tamaño aproximado a una cuarta parte del tamaño del arreglo original, junto con dos llamadas al método `combinar`. Estas dos llamadas al método `combinar` requieren, a lo más, $n/2 - 1$ comparaciones para un número total de $O(n)$ comparaciones. Este proceso continúa, y cada llamada a `ordenarArreglo` genera dos llamadas adicionales a `ordenarArreglo` y una llamada a `combinar`, hasta que el algoritmo divide el arreglo en subarreglos de un elemento. En cada nivel, se requieren $O(n)$ comparaciones para combinar los subarreglos. Cada nivel divide el tamaño de los arreglos a la mitad, por lo que al duplicar el tamaño del arreglo se requiere un nivel más. Si se cuadruplica el tamaño del arreglo, se requieren dos niveles más. Este patrón es logarítmico, y produce $\log_2 n$ niveles. Esto resulta en una eficiencia total de $O(n \log n)$. En la figura 16.12 se sintetizan muchos de los algoritmos de búsqueda y ordenamiento que cubrimos en este libro, y se enlista el valor de Big O para cada uno de ellos. En la figura 16.13 se enlistan los valores de Big O que hemos cubierto en este capítulo, junto con cierto número de valores para *n*, para resaltar las diferencias en las proporciones de crecimiento.

Algoritmo	Ubicación	Big O
Algoritmos de búsqueda:		
Búsqueda lineal	Sección 16.2.1.	$O(n)$
Búsqueda binaria	Sección 16.2.2.	$O(\log n)$
Búsqueda lineal recursiva	Ejercicio 16.8.	$O(n)$
Búsqueda binaria recursiva	Ejercicio 16.9.	$O(\log n)$

Figura 16.12 | Algoritmos de búsqueda y ordenamiento con valores de Big O. (Parte 1 de 2).

Algoritmo	Ubicación	Big O
Algoritmos de ordenamiento:		
Ordenamiento por selección	Sección 16.3.1	$O(n^2)$
Ordenamiento por inserción	Sección 16.3.2	$O(n^2)$
Ordenamiento por combinación	Ejercicio 16.3.3	$O(n \log n)$
Ordenamiento de burbuja	Ejercicios 16.3 y 16.4	$O(n^2)$

Figura 16.12 | Algoritmos de búsqueda y ordenamiento con valores de Big O. (Parte 2 de 2).

n =	$O(\log n)$	$O(n)$	$O(n \log n)$	$O(n^2)$
1	0	1	0	1
2	1	2	2	4
3	1	3	3	9
4	1	4	4	6
5	1	5	5	25
10	1	10	10	100
100	2	100	200	10,000
1000	3	1000	3000	10^6
1,000,000	6	1,000,000	6,000,000	10^{12}
1,000,000,000	9	1,000,000,000	9,000,000,000	10^{18}

Figura 16.13 | Número de comparaciones para las notaciones comunes de Big O.

16.4 Invariantes

Después de escribir una aplicación, un programador comúnmente la prueba a conciencia. Es bastante difícil crear un conjunto exhaustivo de pruebas, y siempre es posible que un caso específico no se evalúe. Una técnica que nos puede ayudar a probar nuestros programas en forma exhaustiva es el uso de las invariantes. Una **invariante** es una aserción (vea la sección 13.13) que es verdadera antes y después de ejecutar una sección del código. Las invariantes son matemáticas por naturaleza, y sus conceptos son más aplicables en el lado teórico de las ciencias computacionales.

El tipo más común de invariante es una **invariante de ciclo**, la cual es una aserción que permanece siendo verdadera

- antes de la ejecución del ciclo,
- después de cada iteración del cuerpo del ciclo, y
- cuando termina el ciclo.

Una invariante de ciclo escrita en forma apropiada nos puede ayudar a codificar un ciclo de manera correcta. Hay cuatro pasos para desarrollar un ciclo a partir de una invariante de ciclo.

1. Establecer los valores iniciales para las variables de control de ciclo.
2. Determinar la condición que hace que el ciclo termine.
3. Modificar la(s) variable(s) de control, de manera que el ciclo progrese hacia su terminación.
4. Comprobar que la invariante siga siendo verdadera al final de cada iteración.

Como ejemplo, examinaremos el método `busquedaLineal` de la clase `ArregloLineal` en la figura 16.2. La invariante para el algoritmo de búsqueda lineal es:

> *para todas las* `k` *tales que* `0 <= k` *y* `k < indice`
> `datos[ k ] != claveBusqueda`

Por ejemplo, suponga que `indice` es igual a 3. Si elegimos cualquier número no negativo menor que 3, como 1 para el valor de `k`, el elemento en `datos` en la ubicación `k` en el arreglo no es igual a `claveBusqueda`. En esencia, esta invariante establece que la porción del arreglo, llamada **subarreglo**, que abarca desde el inicio del arreglo hasta, pero sin incluir el elemento en `indice`, no contiene la `claveBusqueda`. Un subarreglo puede contener cualquier número de elementos.

De acuerdo con el *paso 1*, debemos inicializar primero la variable de control `indice`. De la invariante podemos ver que, si establecemos `indice` en `0`, entonces el subarreglo contiene cero elementos. Por lo tanto, la invariante es verdadera, ya que un subarreglo sin elementos no puede contener un valor que coincida con la `claveBusqueda`.

El segundo paso es determinar la condición que hace que el ciclo termine. El ciclo debe terminar después de buscar en todo el arreglo; cuando `indice` es igual a la longitud del arreglo. En este caso, ningún elemento del arreglo `datos` coincide con la `claveBusqueda`. Una vez que el `indice` llega al final del arreglo, la invariante sigue siendo verdadera; ningún elemento en el subarreglo (que en este caso es todo el arreglo) es igual a la `claveBusqueda`.

Para que el ciclo proceda al siguiente elemento, incrementamos la variable de control `indice`. El último paso es asegurar que la invariante siga siendo verdadera después de cada iteración. La instrucción `if` (líneas 26 y 27 de la figura 16.2) determina si `datos[ indice ]` es igual a la `claveBusqueda`. De ser así, el método termina y devuelve `indice`. Como `indice` es la primera ocurrencia de `claveBusqueda` en `datos`, la invariante sigue siendo verdadera; el subarreglo hasta `indice` no contiene la `claveBusqueda`.

16.5 Conclusión

En este capítulo se presentaron las técnicas de ordenamiento y búsqueda. Hablamos sobre dos algoritmos de búsqueda (la búsqueda lineal y la búsqueda binaria) y tres algoritmos de ordenamiento (el ordenamiento por selección, por inserción y por combinación). Presentamos la notación Big O, la cual nos ayuda a analizar la eficiencia de un algoritmo. También aprendió acerca de las invariantes de ciclo, que deben seguir siendo verdaderas antes de que el ciclo empiece a ejecutarse, mientras se está ejecutando y cuando termine su ejecución. En el siguiente capítulo aprenderá acerca de las estructuras de datos dinámicas, que pueden aumentar o reducir su tamaño en tiempo de ejecución.

Resumen

Sección 16.1 Introducción

- La búsqueda de datos implica determinar si una clave de búsqueda está presente en los datos y, de ser así, encontrar su ubicación.
- El ordenamiento implica poner los datos en orden.

Sección 16.2 Algoritmos de búsqueda

- El algoritmo de búsqueda lineal busca cada elemento en el arreglo en forma secuencial, hasta que encuentra el elemento correcto. Si el elemento no se encuentra en el arreglo, el algoritmo evalúa cada elemento en el arreglo y, cuando llega al final del mismo, informa al usuario que el elemento no está presente. Si el elemento se encuentra en el arreglo, la búsqueda lineal evalúa cada elemento hasta encontrar el correcto.
- Una de las principales diferencias entre los algoritmos de búsqueda es la cantidad de esfuerzo que requieren para poder devolver un resultado.
- Una manera de describir la eficiencia de un algoritmo es mediante la notación Big O, la cual indica qué tan duro tiene que trabajar un algoritmo para resolver un problema.

- Para los algoritmos de búsqueda y ordenamiento, Big O depende a menudo de cuántos elementos haya en los datos.
- Un algoritmo que es $O(1)$ no necesariamente requiere sólo una comparación. Sólo significa que el número de comparaciones no aumenta a medida que se incrementa el tamaño del arreglo.
- Se considera que un algoritmo $O(n)$ tiene un tiempo de ejecución lineal.
- La notación Big O está diseñada para resaltar los factores dominantes, e ignorar los términos que pierden importancia con valores altos de *n*.
- La notación Big O se enfoca en la proporción de crecimiento de los tiempos de ejecución de los algoritmos, por lo que se ignoran las constantes.
- El algoritmo de búsqueda lineal se ejecuta en un tiempo $O(n)$.
- El peor caso en la búsqueda lineal es que se debe comprobar cada elemento para determinar si el elemento de búsqueda existe. Esto ocurre si la clave de búsqueda es el último elemento en el arreglo, o si no está presente.
- El algoritmo de búsqueda binaria es más eficiente que el algoritmo de búsqueda lineal, pero requiere que el arreglo esté ordenado.
- La primera iteración de la búsqueda binaria evalúa el elemento medio del arreglo. Si es igual a la clave de búsqueda, el algoritmo devuelve su ubicación. Si la clave de búsqueda es menor que el elemento medio, la búsqueda binaria continúa con la primera mitad del arreglo. Si la clave de búsqueda es mayor que el elemento medio, la búsqueda binaria continúa con la segunda mitad del arreglo. En cada iteración de la búsqueda binaria se evalúa el valor medio del resto del arreglo y, si no se encuentra el elemento, se elimina la mitad de los elementos restantes.
- La búsqueda binaria es un algoritmo de búsqueda más eficiente que la búsqueda lineal, ya que cada comparación elimina la mitad de los elementos del arreglo a considerar.
- La búsqueda binaria se ejecuta en un tiempo $O(\log n)$, ya que cada paso elimina la mitad de los elementos restantes.
- Si el tamaño del arreglo se duplica, la búsqueda binaria sólo requiere una comparación adicional para completarse con éxito.

Sección 16.3 Algoritmos de ordenamiento

- El ordenamiento por selección es un algoritmo de ordenamiento simple, pero ineficiente.
- En la primera iteración del algoritmo por selección, se selecciona el elemento más pequeño en el arreglo y se intercambia con el primer elemento. En la segunda iteración del ordenamiento por selección, se selecciona el segundo elemento más pequeño (que viene siendo el elemento restante más pequeño) y se intercambia con el segundo elemento. El ordenamiento por selección continúa hasta que en la última iteración se selecciona el segundo elemento más grande, y se intercambia con el antepenúltimo elemento, dejando el elemento más grande en el último índice. En la *i*-ésima iteración del ordenamiento por selección, los *i* elementos más pequeños de todo el arreglo se ordenan en los primeros *i* índices.
- El algoritmo de ordenamiento por selección se ejecuta en un tiempo $O(n^2)$.
- En la primera iteración del ordenamiento por inserción, se toma el segundo elemento en el arreglo y, si es menor que el primer elemento, se intercambian. En la segunda iteración del ordenamiento por inserción, se analiza el tercer elemento y se inserta en la posición correcta, con respecto a los primeros dos elementos. Después de la *i*-ésima iteración del ordenamiento por inserción, quedan ordenados los primeros *i* elementos del arreglo original.
- El algoritmo de ordenamiento por inserción se ejecuta en un tiempo $O(n^2)$.
- El ordenamiento por combinación es un algoritmo de ordenamiento que es más rápido, pero más complejo de implementar, que el ordenamiento por selección y el ordenamiento por inserción.
- Para ordenar un arreglo, el algoritmo de ordenamiento por combinación lo divide en dos subarreglos de igual tamaño, ordena cada subarreglo en forma recursiva y combina los subarreglos en un arreglo más grande.
- El caso base del ordenamiento por combinación es un arreglo con un elemento. Un arreglo de un elemento ya está ordenado, por lo que el ordenamiento por combinación regresa de inmediato, cuando se llama con un arreglo de un elemento. La parte de este algoritmo que corresponde al proceso de combinar recibe dos arreglos ordenados (éstos podrían ser arreglos de un elemento) y los combina en un arreglo ordenado más grande.
- Para realizar la combinación, el ordenamiento por combinación analiza el primer elemento en cada arreglo, que también es el elemento más pequeño en el arreglo. El ordenamiento por combinación recibe el más pequeño de estos elementos y lo coloca en el primer elemento del arreglo más grande. Si aún hay elementos en el subarreglo, el ordenamiento por combinación analiza el segundo elemento en el subarreglo (que ahora es el elemento más pequeño restante) y lo compara con el primer elemento en el otro subarreglo. El ordenamiento por combinación continúa con este proceso, hasta que se llena el arreglo más grande.
- En el peor caso, la primera llamada al ordenamiento por combinación tiene que realizar $O(n)$ comparaciones para llenar las *n* posiciones en el arreglo final.

- La porción del algoritmo de ordenamiento por combinación que corresponde al proceso de combinar se realiza en dos subarreglos, cada uno de un tamaño aproximado a $n/2$. Para crear cada uno de estos subarreglos, se requieren $n/2 - 1$ comparaciones para cada subarreglo, o un total de $O(n)$ comparaciones. Este patrón continúa a medida que cada nivel trabaja hasta en el doble de esa cantidad de arreglos, pero cada uno equivale a la mitad del tamaño del arreglo anterior.
- De manera similar a la búsqueda binaria, esta acción de partir los subarreglos a la mitad produce un total de log n niveles, para una eficiencia total de $O(n \log n)$.

Sección 16.4 Invariantes

- Una invariante es una aserción que es verdadera antes y después de la ejecución de una parte del código de un programa.
- Una invariante de ciclo es una aserción que es verdadera antes de empezar a ejecutar el ciclo, durante cada iteración del mismo y después de que el ciclo termina.

Terminología

Big O, notación	$O(n)$
búsqueda	$O(n^2)$
búsqueda binaria	ordenamiento
búsqueda lineal	ordenamiento por combinación
clave de búsqueda	ordenamiento por inserción
invariante	ordenamiento por selección
invariante de ciclo	tiempo de ejecución constante
$O(1)$	tiempo de ejecución cuadrático
$O(\log n)$	tiempo de ejecución lineal
$O(n \log n)$	tiempo de ejecución logarítmico

Ejercicios de autoevaluación

16.1 Complete los siguientes enunciados:

a) Una aplicación de ordenamiento por selección debe requerir un tiempo aproximado de ____________ veces más para ejecutarse en un arreglo de 128 elementos, en comparación con un arreglo de 32 elementos.

b) La eficiencia del ordenamiento por combinación es de ______________.

16.2 ¿Qué aspecto clave de la búsqueda binaria y del ordenamiento por combinación es responsable de la parte logarítmica de sus respectivos valores Big O?

16.3 ¿En qué sentido es superior el ordenamiento por inserción al ordenamiento por combinación? ¿En qué sentido es superior el ordenamiento por combinación al ordenamiento por inserción?

16.4 En el texto decimos que, una vez que el ordenamiento por combinación divide el arreglo en dos subarreglos, después ordena estos dos subarreglos y los combina. ¿Por qué alguien podría quedar desconcertado al decir nosotros que "después ordena estos dos subarreglos"?

Respuestas a los ejercicios de autoevaluación

16.1 a) 16, ya que un algoritmo $O(n^2)$ requiere 16 veces más de tiempo para ordenar hasta cuatro veces más información. b) $O(n \log n)$.

16.2 Ambos algoritmos incorporan la acción de "dividir a la mitad" (reducir algo de cierta forma a la mitad). La búsqueda binaria elimina del proceso una mitad del arreglo después de cada comparación. El ordenamiento por combinación divide el arreglo a la mitad, cada vez que se llama.

16.3 El ordenamiento por inserción es más fácil de comprender y de programar que el ordenamiento por combinación. El ordenamiento por combinación es mucho más eficiente [$O(n \log n)$] que el ordenamiento por inserción [$O(n^2)$].

16.4 En cierto sentido, en realidad no ordena estos dos subarreglos. Simplemente sigue dividiendo el arreglo original a la mitad, hasta que obtiene un subarreglo de un elemento, que desde luego, está ordenado. Después construye los dos subarreglos originales al combinar estos arreglos de un elemento para formar subarreglos más grandes, los cuales se mezclan, y así en lo sucesivo.

Ejercicios

16.5 *(Ordenamiento de burbuja)* Implemente el ordenamiento de burbuja, otra técnica de ordenamiento simple, pero ineficiente. Se le llama ordenamiento de burbuja o de hundimiento, debido a que los valores más pequeños van "subiendo como burbujas" gradualmente, hasta llegar a la parte superior del arreglo (es decir, hacia el primer elemento) como las burbujas de aire que se elevan en el agua, mientras que los valores más grandes se hunden en el fondo (final) del arreglo. Esta técnica utiliza ciclos anidados para realizar varias pasadas a través del arreglo. Cada pasada compara pares sucesivos de elementos. Si un par se encuentra en orden ascendente (o los valores son iguales), el ordenamiento de burbuja deja los valores como están. Si un par se encuentra en orden descendente, el ordenamiento de burbuja intercambia sus valores en el arreglo.

En la primera pasada se comparan los primeros dos elementos del arreglo, y se intercambian sus valores si es necesario. Después se comparan los elementos segundo y tercero en el arreglo. En la pasada final, se comparan los últimos dos elementos en el arreglo y se intercambian, en caso de ser necesario. Después de una pasada, el elemento más grande estará en el último índice. Después de dos pasadas, los dos elementos más grandes se encontrarán en los últimos dos índices. Explique por qué el ordenamiento de burbuja es un algoritmo $O(n^2)$.

16.6 *(Ordenamiento de burbuja mejorado)* Realice las siguientes modificaciones simples para mejorar el rendimiento del ordenamiento de burbuja que desarrolló en el ejercicio 16.5:

a) Después de la primera pasada, se garantiza que el número más grande estará en el elemento con la numeración más alta del arreglo; después de la segunda pasada, los dos números más altos estarán "acomodados"; y así en lo sucesivo. En vez de realizar nueve comparaciones en cada pasada, modifique el ordenamiento de burbuja para que realice ocho comparaciones en la segunda pasada, siete en la tercera, y así en lo sucesivo.

b) Los datos en el arreglo tal vez se encuentren ya en el orden apropiado, o casi apropiado, así que ¿para qué realizar nueve pasadas, si basta con menos? Modifique el ordenamiento para comprobar al final de cada pasada si se han realizado intercambios. Si no se ha realizado ninguno, los datos ya deben estar en el orden apropiado, por lo que el programa debe terminar. Si se han realizado intercambios, por lo menos, se necesita una pasada más.

16.7 *(Ordenamiento de cubeta)* Un ordenamiento de burbuja comienza con un arreglo unidimensional de enteros positivos que se deben ordenar, y un arreglo bidimensional de enteros, en el que las filas están indexadas de 0 a 9 y las columnas de 0 a $n - 1$, en donde n es el número de valores a ordenar. Cada fila del arreglo bidimensional se conoce como una *cubeta*. Escriba una clase llamada `OrdenamientoCubeta`, que contenga un método llamado `ordenar` y que opere de la siguiente manera:

a) Coloque cada valor del arreglo unidimensional en una fila del arreglo de cubeta, con base en el dígito de las unidades (el de más a la derecha) del valor. Por ejemplo, el número 97 se coloca en la fila 7, el 3 se coloca en la fila 3 y el 100 se coloca en la fila 0. A este procedimiento se le llama *pasada de distribución*.

b) Itere a través del arreglo de cubeta fila por fila, y copie los valores de vuelta al arreglo original. A este procedimiento se le llama *pasada de recopilación*. El nuevo orden de los valores anteriores en el arreglo unidimensional es 100, 3 y 97.

c) Repita este proceso para cada posición de dígito subsiguiente (decenas, centenas, miles, etcétera). En la segunda pasada (el dígito de las decenas) se coloca el 100 en la fila 0, el 3 en la fila 0 (ya que 3 no tiene dígito de decenas) y el 97 en la fila 9. Después de la pasada de recopilación, el orden de los valores en el arreglo unidimensional es 100, 3 y 97. En la tercera pasada (dígito de las centenas), el 100 se coloca en la fila 1, el 3 en la fila 0 y el 97 en la fila 0 (después del 3). Después de esta última pasada de recopilación, el arreglo original se encuentra en orden.

Observe que el arreglo bidimensional de cubetas es 10 veces la longitud del arreglo entero que se está ordenando. Esta técnica de ordenamiento proporciona un mejor rendimiento que el ordenamiento de burbuja, pero requiere mucha más memoria; el ordenamiento de burbuja requiere espacio sólo para un elemento adicional de datos. Esta comparación es un ejemplo de la concesión entre espacio y tiempo: el ordenamiento de cubeta utiliza más memoria que el ordenamiento de burbuja, pero su rendimiento es mejor. Esta versión del ordenamiento de cubeta requiere copiar todos los datos de vuelta al arreglo original en cada pasada. Otra posibilidad es crear un segundo arreglo de cubeta bidimensional, e intercambiar en forma repetida los datos entre los dos arreglos de cubeta.

16.8 *(Búsqueda lineal recursiva)* Modifique la figura 16.2 para utilizar el método recursivo `busquedaLinealRecursiva` para realizar una búsqueda lineal en el arreglo. El método debe recibir la clave de búsqueda y el índice inicial como

argumentos. Si se encuentra la clave de búsqueda, se devuelve su índice en el arreglo; en caso contrario, se devuelve `-1`. Cada llamada al método recursivo debe comprobar un índice en el arreglo.

16.9 *(Búsqueda binaria recursiva)* Modifique la figura 16.4 para que utilice el método recursivo `busquedaBinariaRecursiva` para realizar una búsqueda binaria en el arreglo. El método debe recibir la clave de búsqueda, el índice inicial y el índice final como argumentos. Si se encuentra la clave de búsqueda, se devuelve su índice en el arreglo. Si no se encuentra, se devuelve `-1`.

16.10 *(Quicksort)* La técnica de ordenamiento recursiva llamada "quicksort" utiliza el siguiente algoritmo básico para un arreglo unidimensional de valores:

a) *Paso de particionamiento:* tomar el primer elemento del arreglo desordenado y determinar su ubicación final en el arreglo ordenado (es decir, todos los valores a la izquierda del elemento en el arreglo son menores que el elemento, y todos los valores a la derecha del elemento en el arreglo son mayores; a continuación le mostraremos cómo hacer esto). Ahora tenemos un elemento en su ubicación apropiada y dos subarreglos desordenados.

b) *Paso recursivo:* llevar a cabo el *paso 1* en cada subarreglo desordenado. Cada vez que se realiza el *paso 1* en un subarreglo, se coloca otro elemento en su ubicación final en el arreglo ordenado, y se crean dos subarreglos desordenados. Cuando un subarreglo consiste en un elemento, ese elemento se encuentra en su ubicación final (debido a que un arreglo de un elemento ya está ordenado).

El algoritmo básico parece bastante simple, pero ¿cómo determinamos la posición final del primer elemento de cada subarreglo? Como ejemplo, considere el siguiente conjunto de valores (el elemento en negritas es el elemento de particionamiento; se colocará en su ubicación final en el arreglo ordenado):

37 2 6 4 89 8 10 12 68 45

Empezando desde el elemento de más a la derecha del arreglo, se compara cada elemento con 37 hasta que se encuentra un elemento menor que 37; después se intercambian el 37 y ese elemento. El primer elemento menor que 37 es 12, por lo que se intercambian el 37 y el 12. El nuevo arreglo es

12 2 6 4 89 8 10 **37** 68 45

El elemento 12 está en cursivas, para indicar que se acaba de intercambiar con el 37.

Empezando desde la parte izquierda del arreglo, pero con el elemento que está después de 12, compare cada elemento con 37 hasta encontrar un elemento mayor que 37; después intercambie el 37 y ese elemento. El primer elemento mayor que 37 es 89, por lo que se intercambian el 37 y el 89.
El nuevo arreglo es

12 2 6 4 **37** 8 10 ***89*** 68 45

Empezando desde la derecha, pero con el elemento antes del 89, compare cada elemento con 37 hasta encontrar un elemento menor que 37; después se intercambian el 37 y ese elemento. El primer elemento menor que 37 es 10, por lo que se intercambian 37 y 10. El nuevo arreglo es

12 2 6 4 *10* 8 **37** 89 68 45

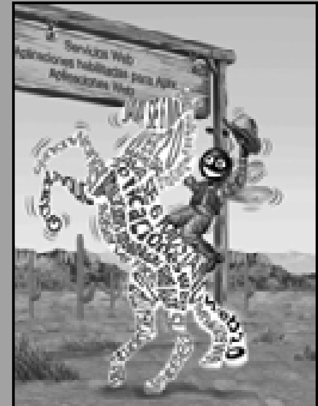
Empezando desde la izquierda, pero con el elemento que está después de 10, compare cada elemento con 37 hasta encontrar un elemento mayor que 37; después intercambie el 37 y ese elemento. No hay más elementos mayores que 37, por lo que al comparar el 37 consigo mismo, sabemos que se ha colocado en su ubicación final en el arreglo ordenado. Cada valor a la izquierda de 37 es más pequeño, y cada valor a la derecha de 37 es más grande.

Una vez que se ha aplicado la partición en el arreglo anterior, hay dos subarreglos desordenados. El subarreglo con valores menores que 37 contiene 12, 2, 6, 4, 10 y 8. El subarreglo con valores mayores que 37 contiene 89, 68 y 45. El ordenamiento continúa en forma recursiva, y ambos subarreglos se particionan de la misma forma que el arreglo original.

Con base en la discusión anterior, escriba el método recursivo `ayudanteQuicksort` para ordenar un arreglo entero unidimensional. El método debe recibir como argumentos un índice inicial y un índice final en el arreglo original que se está ordenando.

17

Estructuras de datos



De muchas cosas a las que estoy atado, no he podido liberarme;
Y muchas de las que me liberé, han vuelto a mí.

—Lee Wilson Dodd

'¿Podría caminar un poco más rápido?' Dijo una merluza a un caracol; 'Hay una marsopa acercándose mucho a nosotros y está pisándome la cola'.

—Lewis Carroll

Siempre hay lugar en la cima.

—Daniel Webster

Empujen; sigan moviéndose.

—Thomas Morton

Daré vuelta a una nueva hoja.

—Miguel de Cervantes

OBJETIVOS

En este capítulo aprenderá a:

- Formar estructuras de datos enlazadas utilizando referencias, clases autorreferenciadas y recursividad.
- Conocer las clases de envoltura de tipos, que permiten a los programas procesar valores de datos primitivos como objetos.
- Utilizar autoboxing para convertir un valor primitivo en un objeto de la clase de envoltura de tipo correspondiente.
- Utilizar autounboxing para convertir un objeto de una clase de envoltura de tipo en un valor primitivo.
- Crear y manipular estructuras dinámicas de datos como listas enlazadas, colas, pilas y árboles binarios.
- Comprender varias aplicaciones importantes de las estructuras de datos enlazadas.
- Crear estructuras de datos reutilizables con clases, herencia y composición.

Plan general

17.1 Introducción

En capítulos anteriores hemos estudiado **estructuras de datos** de tamaño fijo, como los arreglos unidimensionales y multidimensionales. En este capítulo presentaremos las **estructuras de datos dinámicas**, que crecen y se reducen en tiempo de ejecución. Las **listas enlazadas** son colecciones de elementos de datos "alineados en una fila"; pueden insertarse y eliminarse elementos en cualquier parte de una lista enlazada. Las **pilas** son importantes en los compiladores y sistemas operativos; pueden insertarse y eliminarse elementos sólo en un extremo de una pila: su **parte superior**. Las **colas** representan líneas de espera; se insertan elementos en la parte final (conocida como **rabo**) de una cola y se eliminan elementos de su parte inicial (conocida como **cabeza**). Los **árboles binarios** facilitan la búsqueda y ordenamiento de los datos de alta velocidad, la eliminación eficiente de elementos de datos duplicados, la representación de directorios del sistema de archivos, la compilación de expresiones en lenguaje máquina y muchas otras aplicaciones interesantes.

Hablaremos sobre cada uno de estos tipos principales de estructuras de datos e implementaremos programas para crearlas y manipularlas. Utilizaremos clases, herencia y composición para crear y empaquetar estas estructuras de datos, para reutilizarlas y darles mantenimiento. En el capítulo 19, Colecciones, hablaremos sobre las clases predefinidas de Java para implementar las estructuras de datos que describiremos en este capítulo.

Los ejemplos que se presentan aquí son programas prácticos que pueden utilizarse en cursos más avanzados y en aplicaciones industriales. Los ejercicios incluyen una vasta colección de aplicaciones útiles.

Los ejemplos de este capítulo manipulan valores primitivos por cuestión de simpleza. Sin embargo, la mayoría de las implementaciones de estructuras de datos en este capítulo sólo almacenan objetos `Object`. Java tiene una característica conocida como boxing, la cual permite convertir valores primitivos a objetos, y objetos a valores primitivos, para usarlos en casos como éste. Los objetos que representan valores primitivos son instancias de lo que se conoce como clases de envoltura de tipos de Java, en el paquete `java.lang`. En las siguientes dos secciones hablaremos sobre estas clases y las conversiones boxing, para poder utilizarlas en los ejemplos de este capítulo.

Le recomendamos que trate de realizar el proyecto principal descrito en la sección especial titulada Construya su propio compilador. Ha estado utilizando un compilador de Java para traducir sus programas de Java en códigos de bytes, para poder ejecutar estos programas en su computadora. En este proyecto creará su propio compilador funcional. Este compilador leerá un archivo de instrucciones escritas en un lenguaje de alto nivel simple pero poderoso, similar a las primeras versiones del popular lenguaje BASIC. Su compilador traducirá estas instrucciones en un archivo de instrucciones de Lenguaje Máquina Simpletron (LMS); éste es el lenguaje que aprendió en la sección especial del capítulo 7, Construya su propia computadora. ¡Su programa Simulador de Simpletron ejecutará entonces el programa LMS producido por su compilador! Al implementar este proyecto mediante el uso de una metodología orientada a objetos, usted recibirá una maravillosa oportunidad para poner en práctica la mayor parte de lo que ha aprendido en este libro. La sección especial lo lleva cuidadosamente a través de las especificaciones del lenguaje de alto nivel, y describe los algoritmos que usted necesitará para convertir cada instrucción, en lenguaje de alto nivel, a instrucciones de lenguaje máquina. Si disfruta de los retos, tal vez pueda tratar de realizar las diversas mejoras tanto al compilador como al Simulador Simpletron, las cuales se sugieren en los ejercicios.

17.2 Clases de envoltura de tipos para los tipos primitivos

Cada uno de los tipos primitivos (que se enlistan en el apéndice D, Tipos primitivos) tiene su correspondiente **clase de envoltura de tipo** (en el paquete `java.lang`). Estas clases se llaman `Bolean`, `Byte`, `Character`, `Double`, `Float`, `Integer`, `Long` y `Short`. Cada clase de envoltura de tipo nos permite manipular los valores de tipos primitivos como objetos. Muchas de las estructuras de datos que desarrollamos o reutilizamos en los capítulos 17 a 19 manipulan y comparten objetos `Object`. Estas clases no pueden manipular variables de tipos primitivos, pero sí pueden manipular objetos de las clases de envoltura de tipos, ya que en última instancia, cada clase se deriva de `Object`.

Cada una de las clases de envoltura de tipos numéricos (`Byte`, `Short`, `Integer`, `Long`, `Float` y `Double`) extiende a la clase `Number`. Además, las clases de envoltura de tipos son `final`, por lo que no podemos extenderlas.

Los tipos primitivos no tienen métodos, por lo que los métodos relacionados con un tipo primitivo se encuentran en la clase de envoltura de tipo correspondiente (por ejemplo, el método `parseInt`, que convierte un objeto `String` en un valor `int`, se encuentra en la clase `Integer`). Si necesita manipular un valor primitivo en su programa, primero consulte la documentación para las clases de envoltura de tipos; el método que necesite tal vez ya esté declarado.

17.3 Autoboxing y autounboxing

Antes de Java SE 5, si queríamos insertar un valor primitivo en una estructura de datos que pudiera almacenar sólo objetos `Object`, teníamos que crear un nuevo objeto de la clase de envoltura de tipo correspondiente, y después insertar este objeto en la colección. De manera similar, si queríamos obtener un objeto de una clase de envoltura de tipo de una colección y manipular su valor primitivo, teníamos que invocar un método en el objeto para obtener su correspondiente valor de tipo primitivo. Por ejemplo, suponga que desea agregar un `int` a un arreglo que almacene sólo referencias a objetos `Integer`. Antes de Java SE 5, teníamos que "envolver" un valor `int` en un objeto `Integer` antes de agregar el entero al arreglo y "desenvolver" el valor `int` del objeto `Integer` para obtener el valor del arreglo, como en

```
Integer[] arregloEntero = new Integer[ 5 ]; // crea arregloEntero

// asigna Integer 10 a enteroArreglo[ 0 ]
arregloEntero[ 0 ] = new Integer( 10 );

// obtiene valor int de Integer
int valor = arregloEntero[ 0 ].intValue();
```

Observe que el valor primitivo `int 10` se utiliza para inicializar un objeto `Integer`. Esto logra el resultado deseado, pero requiere código adicional y es pesado. Después necesitamos invocar el método `intValue` de la clase `Integer` para obtener el valor `int` en el objeto `Integer`.

Java SE 5 simplificó la conversión entre valores de tipos primitivos y los objetos de envoltura de tipos, al no requerir código adicional de parte del programador. Java SE 5 introdujo dos nuevas conversiones: la conversión boxing y la conversión unboxing. Una **conversión boxing** convierte un valor de un tipo primitivo en un objeto de la clase de envoltura de tipo correspondiente. Una **conversión unboxing** convierte un objeto de una clase de envoltura de tipo en un valor del tipo primitivo correspondiente. Estas conversiones se pueden realizar de manera automática (a lo cual se le conoce como **autoboxing** y **autounboxing**). Por ejemplo, las instrucciones anteriores se pueden reescribir como

```
Integer[] arregloEntero = new Integer[ 5 ]; // crea arregloEntero
arregloEntero[ 0 ] = 10; // asigna Integer 10 a arregloEntero[ 0 ]
int valor = arregloEntero[ 0 ]; // obtiene valor int de Integer
```

En este caso, la conversión autoboxing ocurre cuando se asigna un valor `int` (10) a `arregloEntero[ 0 ]`, ya que `arregloEntero` almacena referencias a objetos `Integer`, no valores primitivos `int`. La conversión autounboxing ocurre cuando se asigna `arregloEntero[ 0 ]` la variable `int valor`, ya que esta variable almacena un valor `int`, no una referencia a un objeto `Integer`. Las conversiones autoboxing y autounboxing también ocurren en las instrucciones de control; la condición de una instrucción de control se puede evaluar como un tipo primitivo `boolean` o un tipo de referencia `Boolean`. Muchos de los ejemplos de este capítulo utilizan estas conversiones para almacenar valores primitivos en, y para obtenerlos de, las estructuras de datos que sólo almacenan referencias a objetos `Object`.

17.4 Clases autorreferenciadas

Una **clase autorreferenciada** contiene una variable de instancia que hace referencia a otro objeto del mismo tipo de clase. Por ejemplo, la declaración:

```
class Nodo
{
    private int datos;
    private Nodo siguienteNodo; // referencia al siguiente nodo enlazado

    public Nodo( int datos ) { /* cuerpo del constructor */ }
    public void establecerDatos( int datos ) { /* cuerpo del método */ }
    public int obtenerDatos() { /* cuerpo del método */ }
    public void establecerSiguiente( Nodo siguiente ) { /* cuerpo del método */ }
    public Nodo obtenerSiguiente() { /* cuerpo del método */ }
} // fin de la clase Nodo
```

declara la clase `Nodo`, la cual tiene dos variables de instancia `private`: la variable entera `datos` y la referencia `Nodo` llamada `siguienteNodo`. El campo `siguienteNodo` hace referencia a un objeto de la clase `Nodo`, un objeto de la misma clase que se está declarando aquí; es por ello que se utiliza el término "clase autorreferenciada". El campo `siguienteNodo` es un **enlace**; "vincula" a un objeto de tipo `Nodo` con otro objeto del mismo tipo. El tipo `Nodo` también tiene cinco métodos: un constructor que recibe un entero para inicializar a `datos`, un método `establecerDatos` para establecer el valor de `datos`, un método `obtenerDatos` para devolver el valor de `datos`, un método `establecerSiguiente` para establecer el valor de `siguienteNodo` y un método `obtenerSiguiente` para devolver una referencia al siguiente nodo.

Los programas pueden enlazar objetos autorreferenciados entre sí para formar estructuras de datos útiles como listas, colas, pilas y árboles. En la figura 17.1 se muestran dos objetos autorreferenciados, enlazados entre sí para formar una lista. Una barra diagonal inversa (que representa una referencia `null`) se coloca en el miembro de enlace del segundo objeto autorreferenciado para indicar que el enlace no hace referencia a otro objeto. La barra diagonal inversa es ilustrativa; no corresponde al carácter de barra diagonal inversa en Java. Utilizamos `null` para indicar el final de una estructura de datos.

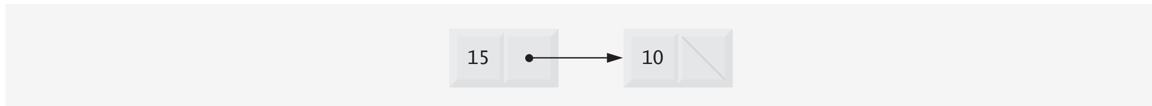


Figura 17.1 | Objetos de una clase autorreferenciada enlazados entre sí.

17.5 Asignación dinámica de memoria

Para crear y mantener estructuras dinámicas de datos se requiere la **asignación dinámica de memoria**; la habilidad para que un programa obtenga más espacio de memoria en tiempo de ejecución, para almacenar nuevos nodos y para liberar el espacio que ya no se necesita. Recuerde que los programas de Java no liberan explícitamente la memoria asignada en forma dinámica. En vez de ello, Java realiza la recolección automática de basura en los objetos que ya no son referenciados en un programa.

El límite para la asignación dinámica de memoria puede ser tan grande como la cantidad de memoria física disponible en la computadora, o la cantidad de espacio en disco disponible en un sistema con memoria virtual. A menudo, los límites son mucho más pequeños ya que la memoria disponible de la computadora debe compartirse entre muchas aplicaciones.

La expresión de declaración y creación de instancia de clase:

```
Nodo nodoParaAgregar = new Nodo( 10 ); // 10 son los datos de nodoParaAgregar
```

asigna la memoria para almacenar un objeto `Nodo` y devuelve una referencia al objeto, que se asigna a `nodoParaAgregar`. Si no hay disponible suficiente memoria, la expresión lanza una excepción `OutOfMemoryError`.

Las siguientes secciones hablan sobre listas, pilas, colas y árboles que utilizan asignación dinámica de memoria y clases autorreferenciadas para crear estructuras de datos dinámicas.

17.6 Listas enlazadas

Una lista enlazada es una colección lineal (es decir, una secuencia) de objetos de una clase autorreferenciada, conocidos como **nodos**, que están conectados por enlaces de referencia; es por ello que se utiliza el término lista "enlazada". Por lo general, un programa accede a una lista enlazada mediante una referencia al primer nodo en la lista. El programa accede a cada nodo subsiguiente a través de la referencia de enlace almacenada en el nodo anterior. Por convención, la referencia de enlace en el último nodo de una lista se establece en `null`. Los datos se almacenan en forma dinámica en una lista enlazada; el programa crea cada nodo según sea necesario. Un nodo puede contener datos de cualquier tipo, incluyendo referencias a objetos de otras clases. Las pilas y las colas son también estructuras de datos lineales y, como veremos, son versiones restringidas de las listas enlazadas. Los árboles son estructuras de datos no lineales.

Pueden almacenarse listas de datos en los arreglos, pero las listas enlazadas ofrecen varias ventajas. Una lista enlazada es apropiada cuando el número de elementos de datos que se van a representar en la estructura de datos es impredecible. Las listas enlazadas son dinámicas, por lo que la longitud de una lista puede incrementarse o reducirse, según sea necesario. Sin embargo, el tamaño de un arreglo "convencional" en Java no puede alterarse; el tamaño del arreglo se fija en el momento en que el programa lo crea. Los arreglos "convencionales" pueden llenarse. Las listas enlazadas se llenan sólo cuando el sistema no tiene suficiente memoria para satisfacer las peticiones de asignación dinámica de almacenamiento. El paquete `java.util` contiene la clase `LinkedList` para implementar y manipular listas enlazadas que crezcan y se reduzcan durante la ejecución del programa. Hablaremos sobre la clase `LinkedList` en el capítulo 19, Colecciones.



Tip de rendimiento 17.1

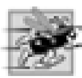
Un arreglo puede declararse de manera que contenga más elementos que el número de elementos esperados, pero esto desperdicia memoria. Las listas enlazadas proporcionan una mejor utilización de memoria en estas situaciones. Las listas enlazadas permiten al programa adaptarse a las necesidades de almacenamiento en tiempo de ejecución.



Tip de rendimiento 17.2

La inserción en una lista enlazada es rápida; sólo hay que modificar dos referencias (después de localizar el punto de inserción). Todos los objetos nodo existentes permanecen en sus posiciones actuales en memoria.

Las listas enlazadas pueden mantenerse en orden con sólo insertar cada nuevo elemento en el punto apropiado de la lista. (Claro que lleva tiempo localizar el punto de inserción apropiado). Los elementos existentes en la lista no necesitan moverse.



Tip de rendimiento 17.3

La inserción y la eliminación en un arreglo ordenado puede llevar mucho tiempo; todos los elementos que van después del elemento insertado o eliminado deben desplazarse apropiadamente.

Por lo general, los nodos de las listas enlazadas no se almacenan contiguamente en memoria, sino que son adyacentes en forma lógica. La figura 17.2 muestra una lista enlazada con varios nodos. Este diagrama presenta una **lista de enlace simple** (cada nodo contiene una referencia al siguiente nodo en la lista). A menudo, las listas

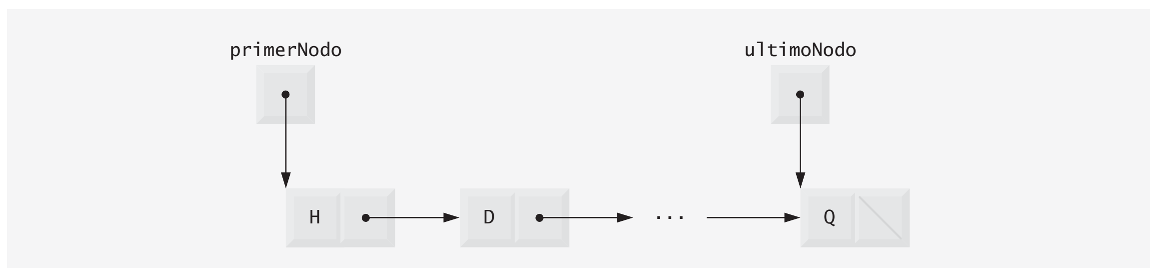


Figura 17.2 | Representación gráfica de una lista enlazada.

enlazadas se implementan como listas de enlace doble (cada nodo contiene una referencia al siguiente nodo en la lista y una referencia al nodo anterior en la lista). La clase `LinkedList` de Java es una implementación de lista de enlace doble.



Tip de rendimiento 17.4

Por lo general, los elementos de un arreglo están contiguos en memoria. Esto permite un acceso inmediato a cualquier elemento del arreglo, ya que la dirección de cualquier elemento puede calcularse directamente como su desplazamiento a partir del inicio del arreglo. Las listas enlazadas no permiten dicho acceso inmediato a sus elementos; para acceder a ellos se tiene que recorrer la lista desde su parte inicial (o desde la parte final en una lista de enlace doble).

El programa de la figuras 17.3 a 17.5 utiliza un objeto de nuestra clase `Lista` para manipular una lista de objetos misceláneos. El programa consta de cuatro clases: `NodoLista` (figura 17.3, líneas 6 a 37), `Lista` (figura 17.3, líneas 40 a 147), `ExcepcionListaVacia` (figura 17.4) y `PruebaLista` (figura 17.5). Las clases `Lista`, `NodoLista` y `ExcepcionListaVacia` están colocadas en el paquete `com.deitel.jhtp7.cap17`, para que puedan reutilizarse a lo largo de este capítulo. Hay una lista enlazada de objetos `NodoLista` encapsulada en cada objeto `Lista`. [*Nota:* muchas de las clases en este capítulo se declaran en el paquete `com.deitel.jhtp7.cap17`. Cada una de estas clases debe compilarse con la opción de línea de comandos `-d` para `javac` Cuando compile las clases que no están en este paquete y cuando ejecute los programas, asegúrese de utilizar la opción `-classpath` para `javac` y `java`, respectivamente].

```
1  // Fig. 17.3: Lista.java
2  // Definiciones de las clases NodoLista y Lista.
3  package com.deitel.jhtp7.cap17;
4
5  // clase para representar un nodo en una lista
6  class NodoLista
7  {
8     // miembros de acceso del paquete; Lista puede acceder a ellos directamente
9     Object datos; // los datos para este nodo
10    NodoLista siguienteNodo; // referencia al siguiente nodo en la lista
11
12    // el constructor crea un objeto NodoLista que hace referencia al objeto
13    NodoLista( Object objeto )
14    {
15       this( objeto, null );
16    } // fin del constructor de NodoLista con un argumento
17
18    // el constructor crea un objeto NodoLista que hace referencia a
19    // un objeto Object y al siguiente objeto NodoLista
20    NodoLista( Object objeto, NodoLista nodo )
21    {
22       datos = objeto;
23       siguienteNodo = nodo;
24    } // fin del constructor de NodoLista con dos argumentos
25
26    // devuelve la referencia a datos en el nodo
27    Object obtenerObject()
28    {
29       return datos; // devuelve el objeto Object en este nodo
30    } // fin del método obtenerObject
31
32    // devuelve la referencia al siguiente nodo en la lista
33    NodoLista obtenerSiguiente()
34    {
35       return siguienteNodo; // obtiene el siguiente nodo
```

Figura 17.3 | Declaraciones de las clases `NodoLista` y `Lista`. (Parte 1 de 3).

```
   } // fin del método obtenerSiguiente
} // fin de la clase NodoLista

// definición de la clase Lista
public class Lista
{
   private NodoLista primerNodo;
   private NodoLista ultimoNodo;
   private String nombre; // cadena como "lista", utilizada para imprimir

   // el constructor crea una Lista vacía con el nombre "lista"
   public Lista()
   {
      this( "lista" );
   } // fin del constructor de Lista sin argumentos

   // el constructor crea una Lista vacía con un nombre
   public Lista( String nombreLista )
   {
      nombre = nombreLista;
      primerNodo = ultimoNodo = null;
   } // fin del constructor de Lista con un argumento

   // inserta objeto Object al frente de la Lista
   public void insertarAlFrente( Object elementoInsertar )
   {
      if ( estaVacia() ) // primerNodo y ultimoNodo hacen referencia al mismo objeto
         primerNodo = ultimoNodo = new NodoLista( elementoInsertar );
      else // primerNodo hace referencia al nuevo nodo
         primerNodo = new NodoLista( elementoInsertar, primerNodo );
   } // fin del método insertarAlFrente

   // inserta objeto Object al final del la Lista
   public void insertarAlFinal( Object elementoInsertar )
   {
      if ( estaVacia() ) // primerNodo y ultimoNodo hacen referencia al mismo objeto
         primerNodo = ultimoNodo = new NodoLista( elementoInsertar );
      else // el siguienteNodo de ultimoNodo hace referencia al nuevo nodo
         ultimoNodo = ultimoNodo.siguienteNodo = new NodoLista( elementoInsertar );
   } // fin del método insertarAlFinal

   // elimina el primer nodo de la Lista
   public Object eliminarDelFrente() throws ExcepcionListaVacia
   {
      if ( estaVacia() ) // lanza excepción si la Lista está vacía
         throw new ExcepcionListaVacia( nombre );

      Object elementoEliminado = primerNodo.datos; // obtiene los datos que se van a eliminar

      // actualiza las referencias primerNodo y ultimoNodo
      if ( primerNodo == ultimoNodo )
         primerNodo = ultimoNodo = null;
      else
         primerNodo = primerNodo.siguienteNodo;

      return elementoEliminado; // devuelve los datos del nodo eliminado
   } // fin del método eliminarDelFrente

```

Figura 17.3 | Declaraciones de las clases `NodoLista` y `Lista`. (Parte 2 de 3).

```
94      // elimina el último nodo de la Lista
95      public Object eliminarDelFinal() throws ExcepcionListaVacia
96      {
97         if ( estaVacia() ) // lanza excepción si la Lista está vacía
98            throw new ExcepcionListaVacia( nombre );
99
100        Object elementoEliminado = ultimoNodo.datos; // obtiene los datos que se van a
                                                       eliminar
101
102        // actualiza las referencias primerNodo y ultimoNodo
103        if ( primerNodo == ultimoNodo )
104           primerNodo = ultimoNodo = null;
105        else // localiza el nuevo último nodo
106        {
107           NodoLista actual = primerNodo;
108
109           // itera mientras el nodo actual no haga referencia a ultimoNodo
110           while ( actual.siguienteNodo != ultimoNodo )
111              actual = actual.siguienteNodo;
112
113           ultimoNodo = actual; // actual el nuevo ultimoNodo
114           actual.siguienteNodo = null;
115        } // fin de else
116
117        return elementoEliminado; // devuelve los datos del nodo eliminado
118     } // fin del método eliminarDelFinal
119
120     // determina si la lista está vacía
121     public boolean estaVacia()
122     {
123        return primerNodo == null; // devuelve true si la lista está vacía
124     } // fin del método estaVacia
125
126     // imprime el contenido de la lista
127     public void imprimir()
128     {
129        if ( estaVacia() )
130        {
131           System.out.printf( "%s vacia\n", nombre );
132           return;
133        } // fin de if
134
135        System.out.printf( "La %s es: ", nombre );
136        NodoLista actual = primerNodo;
137
138        // mientras no esté al final de la lista, imprime los datos del nodo actual
139        while ( actual != null )
140        {
141           System.out.printf( "%s ", actual.datos );
142           actual = actual.siguienteNodo;
143        } // fin de while
144
145        System.out.println( "\n" );
146     } // fin del método imprimir
147  } // fin de la clase Lista
```

Figura 17.3 | Declaraciones de las clases `NodoLista` y `Lista`. (Parte 3 de 3).

La clase `NodoLista` (figura 17.3, líneas 6 a 37) declara los campos de acceso del paquete llamados `datos` y `siguienteNodo`. El campo `datos` es una referencia `Object`, por lo que puede hacer referencia a cualquier objeto.

El miembro `siguienteNodo` de `NodoLista` almacena una referencia al siguiente objeto `NodoLista` en la lista enlazada (o `null`, si el nodo es el último en la lista).

En las líneas 42 y 43 de la clase `Lista` (figura 17.3, líneas 40 a 147) se declaran referencias al primer y último objetos `NodoLista` en un objeto `Lista` (`primerNodo` y `ultimoNodo`, respectivamente). Los constructores (líneas 47 a 50 y 53 a 57) inicializan ambas referencias con null. Los métodos más importantes de la clase `Lista` son `insertarAlFrente` (líneas 60 a 66), `insertarAlFinal` (líneas 69 a 75), `eliminarDelFrente` (líneas 78 a 92) y `eliminarDelFinal` (líneas 95 a 118). El método `estaVacia` (líneas 121 a 124) es un *método predicado*, el cual determina si la lista está vacía (es decir, si la referencia al primer nodo de la lista es `null`). Los métodos predicado generalmente evalúan una condición y no modifican el objeto en el que son llamados. Si la lista está vacía, el método `estaVacia` devuelve `true`; en caso contrario, devuelve `false`. El método `imprimir` (líneas 127 a 146) muestra el contenido de la lista. Después de la figura 17.5 hay una discusión detallada sobre los métodos de `Lista`.

El método `main` de la clase `PruebaLista` (figura 17.5) inserta objetos al principio de la lista utilizando el método `insertarAlFrente`, inserta objetos al final de la lista utilizando el método `insertarAlFinal`, elimina objetos de la parte frontal de la lista utilizando el método `eliminarDelFrente` y elimina objetos de la parte final de la lista utilizando el método `eliminarDelFinal`. Después de cada operación de inserción y eliminación, `PruebaLista` invoca al método de `Lista` llamado `imprimir` para mostrar el contenido actual de la lista. Si se trata de eliminar un elemento de una lista vacía, se lanza una excepción del tipo `ExcepcionListaVacia` (figura 17.4), de manera que las llamadas a los métodos `eliminarDelFrente` y `eliminarDelFinal` se colocan en un bloque `try` que va seguido de un manejador de excepciones apropiado. Observe en las líneas 13, 15, 17 y 19 que la aplicación pasa valores literales `int` primitivos a los métodos `insertarAlFrente` e `insertarAlFinal`, aun cuando cada uno de estos métodos se declaró con un parámetro de tipo `Object` (figura 17.3, líneas 60 y 69). En este caso, la JVM realiza la conversión autobox de cada valor literal a un objeto `Integer`, y ese objeto es el que se inserta en la lista. Desde luego que esto se permite debido a que `Object` es una superclase indirecta de `Integer`.

```java
// Fig. 17.4: ExcepcionListaVacia.java
// Definición de la clase ExcepcionListaVacia.
package com.deitel.jhtp7.cap17;

public class ExcepcionListaVacia extends RuntimeException
{
   // constructor sin argumentos
   public ExcepcionListaVacia()
   {
      this( "Lista" ); // llama al otro constructor de ExcepcionListaVacia
   } // fin del constructor de ExcepcionListaVacia sin argumentos

   // constructor con un argumento
   public ExcepcionListaVacia( String nombre )
   {
      super( nombre + " esta vacia" ); // llama al constructor de la superclase
   } // fin del constructor de ExcepcionListaVacia con un argumento
} // fin de la clase ExcepcionListaVacia
```

Figura 17.4 | Declaración de la clase `ExcepcionListaVacia`.

```java
1  // Fig. 17.5: PruebaLista.java
2  // Clase PruebaLista para demostrar las capacidades de Lista.
3  import com.deitel.jhtp7.cap17.Lista;
4  import com.deitel.jhtp7.cap17.ExcepcionListaVacia;
5
6  public class PruebaLista
7  {
```

Figura 17.5 | Manipulaciones de listas enlazadas. (Parte 1 de 2).

```
    public static void main( String args[] )
    {
       Lista lista = new Lista(); // crea el contenedor de Lista

       // inserta enteros en lista
       lista.insertarAlFrente( -1 );
       lista.imprimir();
       lista.insertarAlFrente( 0 );
       lista.imprimir();
       lista.insertarAlFinal( 1 );
       lista.imprimir();
       lista.insertarAlFinal( 5 );
       lista.imprimir();

       // elimina objetos de lista; imprime después de cada eliminación
       try
       {
          Object objetoEliminado = lista.eliminarDelFrente();
          System.out.printf( "%s eliminado\n", objetoEliminado );
          lista.imprimir();

          objetoEliminado = lista.eliminarDelFrente();
          System.out.printf( "%s eliminado\n", objetoEliminado );
          lista.imprimir();

          objetoEliminado = lista.eliminarDelFinal();
          System.out.printf( "%s eliminado\n", objetoEliminado );
          lista.imprimir();

          objetoEliminado = lista.eliminarDelFinal();
          System.out.printf( "%s eliminado\n", objetoEliminado );
          lista.imprimir();
       } // fin de try
       catch ( ExcepcionListaVacia excepcionListaVacia )
       {
          excepcionListaVacia.printStackTrace();
       } // fin de catch
    } // fin de main
 } // fin de la clase PruebaLista
```

```
La lista es: -1

La lista es: 0 -1

La lista es: 0 -1 1

La lista es: 0 -1 1 5

0 eliminado
La lista es: -1 1 5

-1 eliminado
La lista es: 1 5

5 eliminado
La lista es: 1

1 eliminado
lista vacia
```

Figura 17.5 | Manipulaciones de listas enlazadas. (Parte 2 de 2).

Ahora hablaremos detalladamente sobre cada uno de los métodos de la clase `Lista` (figura 17.3) y proporcionaremos diagramas que muestren las manipulaciones de referencia realizadas por los métodos `insertarAlFrente`, `insertarAlFinal`, `eliminarDelFrente` y `eliminarDelFinal`. El método `insertarAlFrente` (líneas 60 a 66 de la figura 17.3) coloca un nuevo nodo al frente de la lista. Los pasos son:

1. Llamar a `estaVacia` para determinar si la lista está vacía (línea 62).
2. Si la lista está vacía, asignar `primerNodo` y `ultimoNodo` al nuevo `NodoLista` que se inicializó con `elementoInsertar` (línea 63). El constructor de `NodoLista` en las líneas 13 a 16 llama al constructor de `NodoLista` en las líneas 20 a 24 para establecer la variable de instancia `datos`, para hacer referencia al objeto `elementoInsertar` que se pasa como argumento y para establecer la referencia `siguienteNodo` en `null`, ya que éste es el primer y último nodo en la lista.
3. Si la lista no está vacía, el nuevo nodo se "enlaza" en la lista asignando a `primerNodo` un nuevo objeto `NodoLista`, e inicializando ese objeto con `elementoInsertar` y `primerNodo` (línea 65). Cuando se ejecuta el constructor de `NodoLista` (líneas 20 a 24), establece la variable de instancia `datos` para que haga referencia al `elementoInsertar` que se pasa como argumento, y realiza la inserción asignando a la referencia `siguienteNodo` del nuevo nodo al objeto `NodoLista` que se pasa como argumento, y que anteriormente era el primer nodo.

En la figura 17.6, la parte (a) muestra una lista y un nuevo nodo durante la operación `insertarAlFrente` y antes de que el programa enlace el nuevo nodo a la lista. Las flechas punteadas en la parte (b) ilustran el *paso 3* de la operación `insertarAlFrente`, en donde se permite al nodo que contiene `12` convertirse en el primer nuevo nodo en la lista.

El método `insertarAlFinal` (líneas 69 a 75 de la figura 17.3) coloca un nuevo nodo al final de la lista. Los pasos son:

1. Llamar a `estaVacia` para determinar si la lista está vacía (línea 71).
2. Si la lista está vacía, asignar `primerNodo` y `ultimoNodo` al nuevo `NodoLista` que se inicializó con `elementoInsertar` (línea 72). El constructor de `NodoLista` en las líneas 13 a 16 llama al constructor de las líneas 20 a 24 para establecer la variable de instancia `datos`, para hacer referencia al objeto `elementoInsertar` que se pasa como argumento y para establecer la referencia `siguienteNodo` en `null`.
3. Si la lista no está vacía, en la línea 74 se enlaza el nuevo nodo a la lista, asignando a `ultimoNodo` y a `ultimoNodo.siguienteNodo` la referencia al nuevo `NodoLista` que se inicializó con `elementoInsertar`. El constructor de `NodoLista` (líneas 13 a 16) establece la variable de instancia `datos` para hacer referencia al objeto `elementoInsertar` que se pasa como argumento y establece la referencia `siguienteNodo` en `null`, ya que éste es el último nodo en la lista.

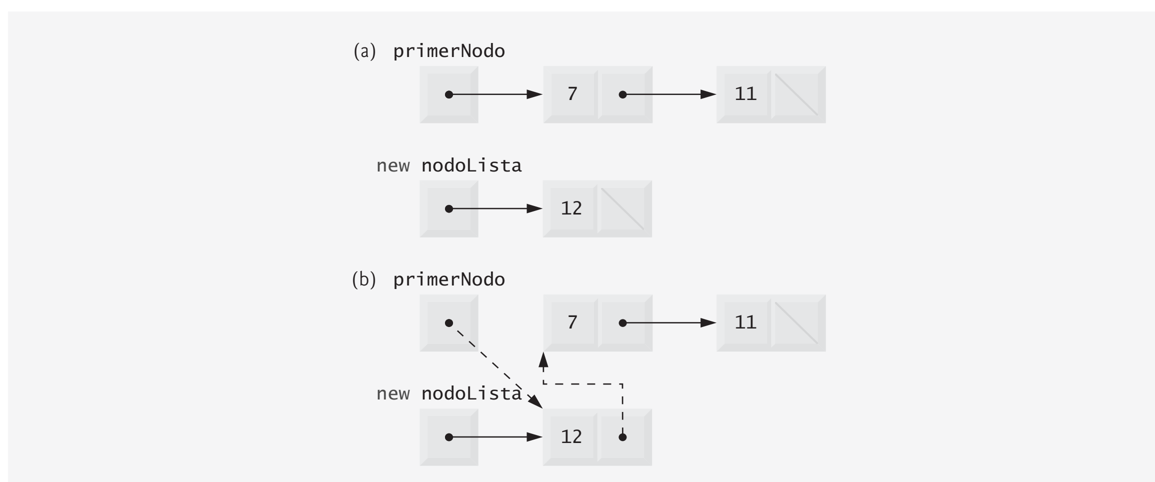


Figura 17.6 | Representación gráfica de la operación `insertarAlFrente`.

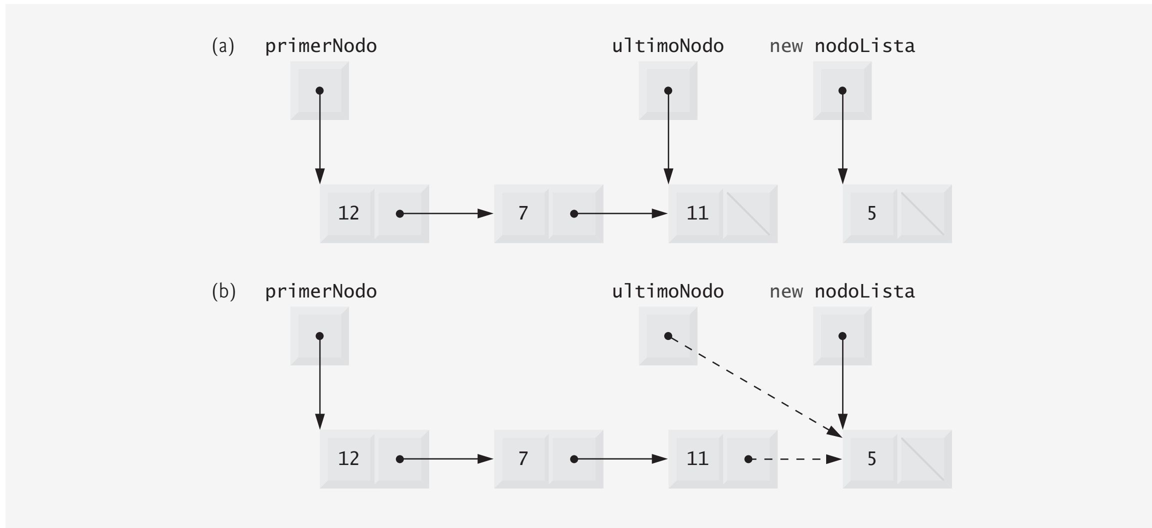


Figura 17.7 | Representación gráfica de la operación `insertarAlFinal`.

En la figura 17.7, la parte (a) muestra una lista y un nuevo nodo durante la operación `insertarAlFrente` y antes de que el programa enlace el nuevo nodo a la lista. Las flechas punteadas en la parte (b) ilustran el *paso 3* del método `insertarAlFinal`, el cual agrega el nuevo nodo al final de una lista que no está vacía.

El método `eliminarDelFrente` (líneas 78 a 92 de la figura 17.3) elimina el primer nodo de la lista y devuelve una referencia a los datos eliminados. El método lanza una excepción `ExcepcionListaVacia` (líneas 80 y 81) si la lista está vacía cuando el programa llama a este método. De no ser así, el método devuelve una referencia a los datos eliminados. Los pasos son:

1. Asignar `primerNodo.datos` (los datos que se van a eliminar de la lista) a la referencia `elementoEliminado` (línea 83).
2. Si `primerNodo` y `ultimoNodo` hacen referencia al mismo objeto (línea 86), quiere decir que la lista sólo tiene un elemento en ese momento. Por lo tanto, el método establece a `primerNodo` y `ultimoNodo` en `null` (línea 87) para eliminar el nodo de la lista (dejándola vacía).
3. Si la lista tiene más de un nodo, entonces el método deja la referencia `ultimoNodo` como está y asigna el valor de `primerNodo.siguienteNodo` a `primerNodo` (línea 89). Por lo tanto, `primerNodo` hace referencia al nodo que era anteriormente el segundo nodo en la lista.
4. Devolver la referencia `elementoEliminado` (línea 91).

En la figura 17.8, la parte (a) ilustra la lista antes de la operación de eliminación. Las líneas punteadas y las flechas en la parte (b) muestran las manipulaciones de referencias.

El método `eliminarDelFinal` (líneas 95 a 118 de la figura 17.3) elimina el último nodo de una lista y devuelve una referencia a los datos eliminados. El método lanza una excepción `ExcepcionListaVacia` (líneas 97 y 98) si la lista está vacía cuando el programa llama a este método. Los pasos son:

1. Asignar `ultimoNodo.datos` (los datos que se van a eliminar de la lista) a `elementoEliminado` (línea 100).
2. Si `primerNodo` y `ultimoNodo` hacen referencia al mismo objeto (línea 103), quiere decir que la lista sólo tiene un elemento en ese momento. Por lo tanto, en la línea 104 se establece a `primerNodo` y `ultimoNodo` en `null` para eliminar ese nodo de la lista (dejándola vacía).
3. Si la lista tiene más de un nodo, crear la referencia `NodoLista` llamada `actual` y asignarla a `primerNodo` (línea 107).

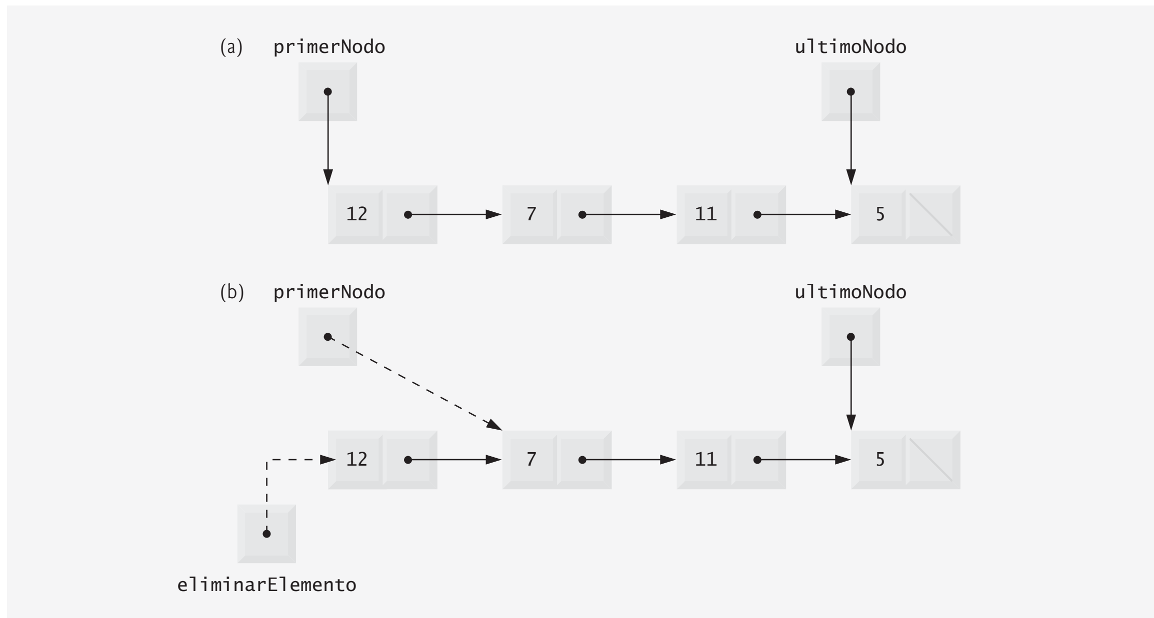


Figura 17.8 | Representación gráfica de la operación `eliminarDelFrente`.

4. Ahora hay que “recorrer la lista” con `actual` hasta que haga referencia al nodo que esté antes del último nodo. El ciclo `while` (líneas 110 y 111) asigna `actual.siguienteNodo` a `actual`, siempre y cuando `actual.siguienteNodo` (el siguiente nodo en la lista) no sea `ultimoNodo`.
5. Después de localizar el penúltimo nodo, asignar `actual` a `ultimoNodo` (línea 113) para actualizar cuál nodo es el último en la lista.
6. Establecer `actual.siguienteNodo` en `null` (línea 114) para eliminar el último nodo de la lista y terminarla con el nodo actual.
7. Devolver la referencia `elementoEliminado` (línea 117).

En la figura 17.9, la parte (a) ilustra la lista antes de la operación de eliminación. Las líneas punteadas y las flechas en la parte (b) muestran las manipulaciones de referencias.

El método `imprimir` (líneas 127 a 146) determina primero si la lista está vacía (líneas 129 a 133). De ser así, `imprimir` muestra un mensaje indicando que la lista está vacía y devuelve el control al método que hizo la llamada. En caso contrario, `imprimir` muestra en pantalla los datos en la lista. En la línea 136 se crea la referencia `NodoLista` llamada `actual` y se inicializa con `primerNodo`. Mientras que `actual` no sea `null`, hay más elementos en la lista. Por lo tanto, en la línea 141 se muestra en pantalla una representación de cadena de `actual.datos`. En la línea 142 se avanza al siguiente nodo en la lista mediante la asignación del valor de la referencia `actual.siguienteNodo` a `actual`. Este algoritmo de impresión es idéntico para listas enlazadas, pilas y colas.

17.7 Pilas

Una pila es una versión restringida de una lista enlazada; pueden agregarse y eliminarse nuevos nodos en una pila solamente desde su parte superior. [*Nota:* una pila no tiene que implementarse mediante el uso de una lista enlazada]. Por esta razón, a una pila se le conoce como **estructura de datos UEPS (último en entrar, primero en salir)**. El miembro de enlace en el nodo inferior (es decir, el último) de la pila se establece en `null` para indicar el fondo de la pila.

Los métodos básicos para manipular una pila son **`push`** (empujar) y **`pop`** (sacar). El método `push` agrega un nuevo nodo a la parte superior de la pila. El método `pop` elimina un nodo de la parte superior de la pila y devuelve los datos del nodo que se quitó.

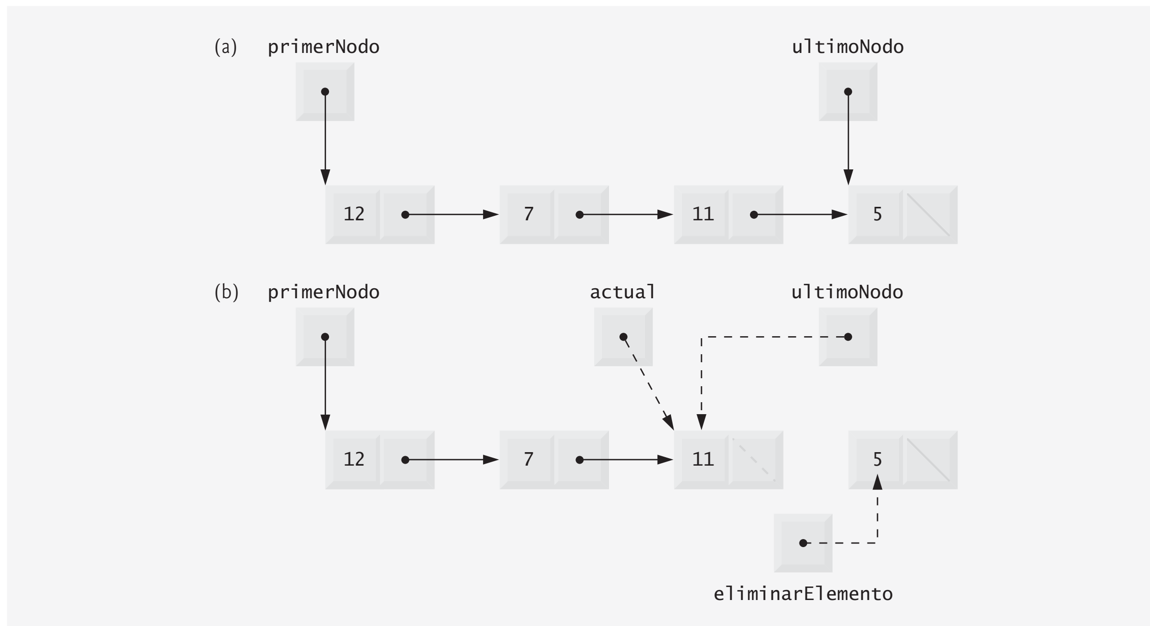


Figura 17.9 | Representación gráfica de la operación `eliminarDelFinal`.

Las pilas tienen muchas aplicaciones interesantes. Por ejemplo, cuando un programa llama a un método, el método llamado debe saber cómo regresar a su invocador, por lo que la dirección de retorno del método que hizo la llamada se mete en la **pila de ejecución del programa**. Si ocurre una serie de llamadas a métodos, las direcciones de retorno sucesivas se meten en la pila en el orden "último en entrar, primero en salir", para que cada método pueda regresar a su invocador. Las pilas soportan llamadas recursivas a métodos de la misma manera que para las llamadas no recursivas convencionales.

La pila de ejecución del programa también contiene la memoria para las variables locales en cada invocación de un método, durante la ejecución de un programa. Cuando el método regresa a su invocador, la memoria para las variables locales de ese método se saca de la pila y esas variables dejan de ser conocidas por el programa. Si la variable local es una referencia, la cuenta de referencias para el objeto al que se refiere se decrementa en 1. Si la cuenta de referencias se vuelve cero, el objeto puede ser candidato para la recolección de basura.

Los compiladores utilizan pilas para evaluar expresiones aritméticas y generar código en lenguaje máquina para procesar las expresiones. Los ejercicios en este capítulo exploran varias aplicaciones de las pilas, incluyendo el utilizarlas para desarrollar un compilador funcional completo. Además, el paquete `java.util` contiene la clase `Stack` (vea el capítulo 19, Colecciones) para implementar y manipular pilas que pueden crecer y reducirse en tamaño durante la ejecución del programa.

Tomaremos ventaja de la estrecha relación entre las listas y las pilas para implementar una clase de pila, mediante la reutilización de una clase de lista. Mostraremos dos formas distintas de reutilización: primero implementaremos la clase de pila extendiendo a la clase `Lista` de la figura 17.3. Después implementaremos una clase de pila con la misma funcionalidad por medio de la composición, incluyendo una referencia a un objeto `Lista` como una variable de instancia privada de una clase de pila. Las estructuras de datos lista, pila y cola en este capítulo se implementan para almacenar referencias `Object`, para exhortar a que se reutilicen en el futuro. Así, puede guardarse cualquier tipo de objeto en una lista, pila o cola.

Clase de pila que hereda de `Lista`

En la aplicación de las figuras 17.10 y 17.11 se crea una clase de pila extendiendo a la clase `Lista` de la figura 17.3. Queremos que la pila tenga los métodos `push`, `pop`, `estaVacia` e `imprimir`. En esencia, éstos son los métodos `insertarAlFrente`, `eliminarDelFrente`, `estaVacia` e `imprimir` de la clase `Lista`. Desde luego que la clase `Lista` contiene otros métodos (como `insertarAlFinal` y `eliminarDelFinal`) que preferiríamos no estuvieran accesibles mediante la interfaz `public` para la clase de pila. Es importante recordar que todos los métodos

en la interfaz `public` de la clase `Lista` también son métodos `public` de la subclase `HerenciaPila` (figura 17.10). Para implementar los métodos de la pila, haremos que cada método de HerenciaPila llame al método apropiado de Lista; el método push llama a `insertarAlFrente` y el método `pop` llama a `eliminarDelFrente`. Los clientes de la clase `HerenciaPila` pueden llamar a los métodos `estaVacia` e `imprimir`, ya que son heredados de `Lista`. La clase `HerenciaPila` se declara como parte del paquete `com.deitel.jhtp7.cap17` para fines de reutilización. Observe que `HerenciaPila` no importa a `Lista`, ya que ambas clases se encuentran en el mismo paquete.

```java
// Fig. 17.10: HerenciaPila.java
// Se deriva de la clase Lista.
package com.deitel.jhtp7.cap17;

public class HerenciaPila extends Lista
{
   // constructor sin argumentos
   public HerenciaPila()
   {
      super( "pila" );
   } // fin del constructor de HerenciaPila sin argumentos

   // agrega objeto a la pila
   public void push( Object objeto )
   {
      insertarAlFrente( objeto );
   } // fin del método push

   // elimina objeto de la pila
   public Object pop() throws ExcepcionListaVacia
   {
      return eliminarDelFrente();
   } // fin del método pop
} // fin de la clase HerenciaPila
```

Figura 17.10 | `HerenciaPila` extiende a la clase `Lista`.

El método `main` de la clase `PruebaHerenciaPila` (figura 17.11) crea un objeto de la clase `HerenciaPila` llamado `pila` (línea 10). El programa mete enteros en la pila (líneas 13, 15, 17 y 19). Observe que, una vez más, se utiliza aquí la conversión autoboxing para insertar objetos `Integer` en la estructura de datos. En las líneas 27 a 32 se sacan objetos de la pila en un ciclo `while` infinito. Si el método `pop` se invoca en una pila vacía, el método lanza una excepción `ExcepcionListaVacia`. En este caso, el programa muestra el rastreo de la pila de la excepción, que muestra los métodos en la pila de ejecución del programa al momento en que ocurrió la excepción. Observe que el programa utiliza el método imprimir (heredado de `Lista`) para mostrar el contenido de la pila.

```java
1  // Fig. 17.11: PruebaHerenciaPila.java
2  // La clase PruebaHerenciaPila.
3  import com.deitel.jhtp7.cap17.HerenciaPila;
4  import com.deitel.jhtp7.cap17.ExcepcionListaVacia;
5
6  public class PruebaHerenciaPila
7  {
8     public static void main( String args[] )
9     {
10       HerenciaPila pila = new HerenciaPila();
11
```

Figura 17.11 | Programa para manipular pilas. (Parte 1 de 2).

```
      // usa el método push
      pila.push( -1 );
      pila.imprimir();
      pila.push( 0 );
      pila.imprimir();
      pila.push( 1 );
      pila.imprimir();
      pila.push( 5 );
      pila.imprimir();

      // elimina elementos de la pila
      try
      {
         Object objetoEliminado = null;

         while ( true )
         {
            objetoEliminado = pila.pop(); // usa el método pop
            System.out.printf( "Se saco %s\n", objetoEliminado );
            pila.imprimir();
         } // fin de while
      } // fin de try
      catch ( ExcepcionListaVacia excepcionListaVacia )
      {
         excepcionListaVacia.printStackTrace();
      } // fin de catch
   } // fin de main
} // fin de la clase PruebaHerenciaPila
```

```
La pila es: -1

La pila es: 0 -1

La pila es: 1 0 -1

La pila es: 5 1 0 -1

Se saco 5
La pila es: 1 0 -1

Se saco 1
La pila es: 0 -1

Se saco 0
La pila es: -1

Se saco -1
pila vacia
com.deitel.jhtp7.cap17.ExcepcionListaVacia: pila esta vacia
        at com.deitel.jhtp7.cap17.Lista.eliminarDelFrente(Lista.java:81)
        at com.deitel.jhtp7.cap17.HerenciaPila.pop(HerenciaPila.java:22)
        at PruebaHerenciaPila.main(PruebaHerenciaPila.java:29)
```

Figura 17.11 | Programa para manipular pilas. (Parte 2 de 2).

Clase de pila que contiene una referencia a una `Lista`

También podemos implementar una clase de pila al reutilizar una clase de lista mediante la composición. En la figura 17.12 se utiliza un objeto `private Lista` (línea 7) en la declaración de la clase `ComposicionPila`. La composición nos permite ocultar los métodos de la clase `Lista` que no deben estar en la interfaz `public` de

```
// Fig. 17.12: ComposicionPila.java
// Definición de la clase ComposicionPila con un objeto Lista compuesto.
package com.deitel.jhtp7.cap17;

public class ComposicionPila
{
   private Lista listaPila;

   // constructor sin argumentos
   public ComposicionPila()
   {
      listaPila = new Lista( "pila" );
   } // fin del constructor de ComposicionPila sin argumentos

   // agrega objeto a la pila
   public void push( Object objeto )
   {
      listaPila.insertarAlFrente( objeto );
   } // fin del método push

   // elimina objeto de la pila
   public Object pop() throws ExcepcionListaVacia
   {
      return listaPila.eliminarDelFrente();
   } // fin del método pop

   // determina si la pila está vacía
   public boolean estaVacia()
   {
      return listaPila.estaVacia();
   } // fin del método estaVacia

   // imprime el contenido de la pila
   public void imprimir()
   {
      listaPila.imprimir();
   } // fin del método imprimir
} // fin de la clase ComposicionPila
```

Figura 17.12 | `ComposicionPila` utiliza un objeto `Lista` compuesto.

nuestra pila. Proporcionamos métodos de interfaz `public` que utilizan solamente los métodos requeridos de `Lista`. Esta técnica de implementar cada método de la pila como una llamada a un método de `Lista` se conoce como `delegación`; el método invocado de la pila `delega` la llamada al método apropiado de `Lista`. Específicamente, `ComposicionPila` delega las llamadas a los métodos de `Lista` `insertarAlFrente`, `eliminarDelFrente`, `estaVacia` e `imprimir`. En este ejemplo no mostramos la clase `PruebaComposicionPila`, ya que la única diferencia en este ejemplo es que cambiamos el tipo de la pila, de `HerenciaPila` a `ComposicionPila` (líneas 3 y 10 de la figura 17.11). La salida es idéntica, utilizando cualquier versión de la pila.

17.8 Colas

Otra estructura de datos que se utiliza comúnmente es la cola. Una cola es similar a la fila para pagar en un supermercado: el cajero atiende primero a la persona que se encuentra hasta adelante. Los demás clientes entran a la fila sólo por su parte final y esperan a que se les atienda. Los nodos de una cola se eliminan sólo desde el principio (cabeza) de la misma y se insertan sólo al final (cola) de ésta. Por esta razón, a una cola se le conoce como estructura de datos **PEPS (primero en entrar, primero en salir)**. Las operaciones para insertar y eliminar se conocen como **enqueue** (agregar a la cola) y **dequeue** (retirar de la cola).

Las colas tienen muchas aplicaciones en los sistemas computacionales. La mayoría de las computadoras tienen sólo un procesador, por lo que sólo pueden atender a una aplicación a la vez. Cada aplicación que requiere tiempo del procesador se coloca en una cola. La aplicación al frente de la cola es la siguiente que recibe atención. Cada aplicación avanza gradualmente al frente de la cola, a medida que las aplicaciones al frente reciben atención.

Las colas también se utilizan para dar soporte al uso de la **cola de impresión**. Por ejemplo, una sola impresora puede compartirse entre todos los usuarios de la red. Muchos usuarios pueden enviar trabajos a la impresora, incluso cuando ésta ya se encuentre ocupada. Estos trabajos de impresión se colocan en una cola hasta que la impresora esté disponible. Un programa conocido como **spooler** administra la cola para asegurarse que, a medida que se complete cada trabajo de impresión, se envíe el siguiente trabajo a la impresora.

En las redes computacionales, los paquetes de información también esperan en colas. Cada vez que un paquete llega a un nodo de la red, debe enrutarse hacia el siguiente nodo en la red a través de la ruta hacia el destino final del paquete. El nodo enrutador envía un paquete a la vez, por lo que los paquetes adicionales se ponen en una cola hasta que el enrutador pueda enviarlos.

Un servidor de archivos en una red computacional se encarga de las peticiones de acceso a los archivos de muchos clientes distribuidos en la red. Los servidores tienen una capacidad limitada para dar servicio a las peticiones de los clientes. Cuando se excede esa capacidad, las peticiones de los clientes esperan en colas.

En la figura 17.13 se crea una clase de cola que contiene un objeto de la clase `Lista` (figura 17.3). La clase `Cola` (figura 17.13) proporciona los métodos `enqueue`, `dequeue`, `estaVacia` e `imprimir`. La clase `Lista` contiene otros métodos (por ejemplo, `insertarAlFrente` y `eliminarDelFrente`) que preferiríamos no estuvieran accesibles mediante la interfaz pública para la clase `Cola`. Mediante el uso de la composición, estos métodos en la interfaz pública de la clase `Lista` no son accesibles para los clientes de la clase `Cola`. Cada método de la clase `Cola` llama a un método apropiado de `Lista`; el método `enqueue` llama al método `insertarAlFinal` de `Lista`, el método `dequeue` llama al método `eliminarDelFrente` de `Lista`, el método `estaVacia` llama al método `estaVacia` de `Lista` y el método `imprimir` llama al método `imprimir` de `Lista`. Para fines de reutilización, la clase `Cola` se declara en el paquete `com.deitel.jhtp7.cap17`.

```java
// Fig. 17.13: Cola.java
// La clase Cola.
package com.deitel.jhtp7.cap17;

public class Cola
{
   private Lista listaCola;

   // constructor sin argumentos
   public Cola()
   {
      listaCola = new Lista( "cola" );
   } // fin del constructor de Cola sin argumentos

   // agrega objeto a la cola
   public void enqueue( Object objeto )
   {
      listaCola.insertarAlFinal( objeto );
   } // fin del método enqueue

   // elimina objeto de la cola
   public Object dequeue() throws ExcepcionListaVacia
   {
      return listaCola.eliminarDelFrente();
   } // fin del método dequeue

   // determina si la cola está vacía
   public boolean estaVacia()
```

Figura 17.13 | `Cola` utiliza la clase `Lista`. (Parte 1 de 2).

```
   {
      return listaCola.estaVacia();
   } // fin del método estaVacia

   // imprime el contenido de la cola
   public void imprimir()
   {
      listaCola.imprimir();
   } // fin del método imprimir
} // fin de la clase Cola
```

Figura 17.13 | `Cola` utiliza la clase `Lista`. (Parte 2 de 2).

El método `main` de la clase `PruebaCola` (figura 17.14) crea un objeto de la clase `Cola` llamado `cola`. En las líneas 13, 15, 17 y 19 se agregan a la cola cuatro enteros, aprovechando la conversión autoboxing para insertar objetos `Integer` en la cola. En las líneas 27 a 32 se utiliza un ciclo `while` infinito para sacar de la cola los objetos, en el orden “primero en entrar, primero en salir”. Cuando la cola está vacía, el método `dequeue` lanza una excepción `ExcepcionListaVacia` y el programa muestra el rastreo de la pila para esa excepción.

```
1  // Fig. 17.14: PruebaCola.java
2  // La clase PruebaCola.
3  import com.deitel.jhtp7.cap17.Cola;
4  import com.deitel.jhtp7.cap17.ExcepcionListaVacia;
5
6  public class PruebaCola
7  {
8     public static void main( String args[] )
9     {
10       Cola cola = new Cola();
11
12       // usa el método enqueue
13       cola.enqueue( -1 );
14       cola.imprimir();
15       cola.enqueue( 0 );
16       cola.imprimir();
17       cola.enqueue( 1 );
18       cola.imprimir();
19       cola.enqueue( 5 );
20       cola.imprimir();
21
22       // elimina objetos de la col
23       try
24       {
25          Object objetoEliminado = null;
26
27          while ( true )
28          {
29             objetoEliminado = cola.dequeue(); // usa el método dequeue
30             System.out.printf( "%s se elimino de la cola\n", objetoEliminado );
31             cola.imprimir();
32          } // fin de while
33       } // fin de try
34       catch ( ExcepcionListaVacia excepcionListaVacia )
35       {
36          excepcionListaVacia.printStackTrace();
```

Figura 17.14 | Programa para procesar objetos `Cola`. (Parte 1 de 2).

```
      } // fin de catch
   } // fin de main
} // fin de la clase PruebaCola
```

```
La cola es: -1

La cola es: -1 0

La cola es: -1 0 1

La cola es: -1 0 1 5

-1 se elimino de la cola
La cola es: 0 1 5

0 se elimino de la cola
La cola es: 1 5

1 se elimino de la cola
La cola es: 5

5 se elimino de la cola
cola vacia
com.deitel.jhtp7.cap17.ExcepcionListaVacia: cola esta vacia
        at com.deitel.jhtp7.cap17.Lista.eliminarDelFrente(Lista.java:81)
        at com.deitel.jhtp7.cap17.Cola.dequeue(Cola.java:24)
        at PruebaCola.main(PruebaCola.java:29)
```

Figura 17.14 | Programa para procesar objetos `Cola`. (Parte 2 de 2).

17.9 Árboles

Las listas enlazadas, pilas y colas son **estructuras de datos lineales** (es decir, **secuencias**). Un árbol es una estructura de datos bidimensional no lineal, con propiedades especiales. Los nodos de un árbol contienen dos o más enlaces. En esta sección hablaremos sobre los árboles binarios (figura 17.15): los árboles cuyos nodos contienen dos enlaces (uno de los cuales puede ser `null`). El **nodo raíz** es el primer nodo en un árbol. Cada enlace en el nodo raíz hace referencia a un **hijo**. El **hijo izquierdo** es el primer nodo en el **subárbol izquierdo** (también conocido como el nodo raíz del subárbol izquierdo), y el **hijo derecho** es el primer nodo en el **subárbol derecho** (también conocido como el nodo raíz del subárbol derecho). Los hijos de un nodo específico se llaman **hermanos**. Un nodo sin hijos se llama **nodo hoja**. Generalmente, los científicos computacionales dibujan árboles desde el nodo raíz hacia abajo; exactamente lo opuesto a la manera en que crecen los árboles naturales.

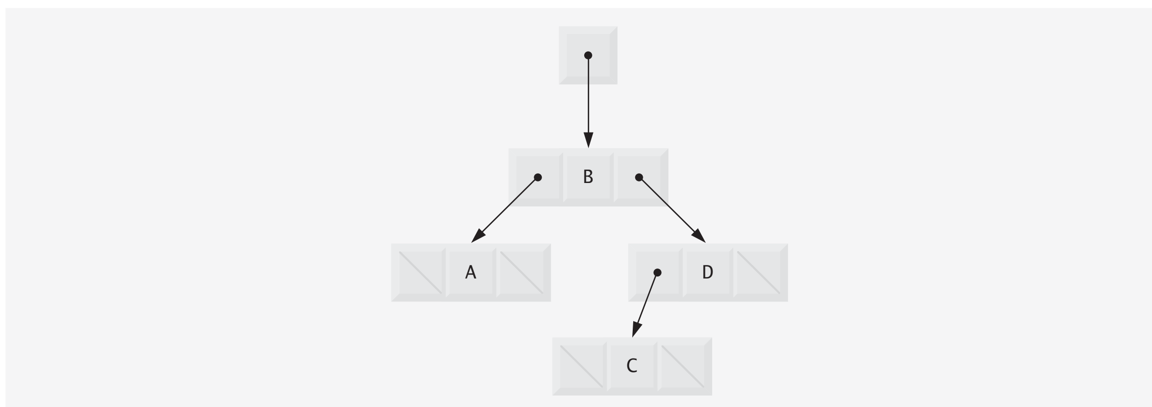


Figura 17.15 | Representación gráfica de un árbol binario.

En nuestro ejemplo de árbol binario crearemos un árbol binario especial, conocido como **árbol de búsqueda binaria**. Un árbol de búsqueda binaria (sin valores de nodo duplicados) cuenta con la característica de que los valores en cualquier subárbol izquierdo son menores que el valor del nodo padre de ese subárbol, y los valores en cualquier subárbol derecho son mayores que el valor del nodo padre de ese subárbol. En la figura 17.16 se muestra un árbol de búsqueda binaria con 12 valores enteros. Observe que la forma del árbol de búsqueda binaria que corresponde a un conjunto de datos puede variar, dependiendo del orden en el que se inserten los valores en el árbol.

La aplicación de las figuras 17.17 y 17.18 crea un árbol de búsqueda binaria compuesto por valores enteros, y lo recorre (es decir, avanza a través de todos sus nodos) de tres maneras: usando los **recorridos inorden**, **preorden** y **postorden** recursivos. El programa genera 10 números aleatorios e inserta a cada uno de ellos en el árbol. La clase `Arbol` se declara en el paquete `com.deitel.jhtp7.cap17` para fines de reutilización.

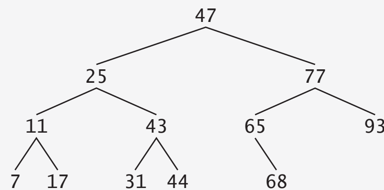


Figura 17.16 | Árbol de búsqueda binario que contiene 12 valores.

```
// Fig. 17.17: Arbol.java
// Definición de las clases NodoArbol y Arbol.
package com.deitel.jhtp7.cap17;

// definición de la clase NodoArbol
class NodoArbol
{
   // miembros de acceso del paquete
   NodoArbol nodoIzq; // nodo izquierdo
   int datos; // valor del nodo
   NodoArbol nodoDer; // nodo derecho

   // el constructor inicializa los datos y hace de este nodo un nodo raíz
   public NodoArbol( int datosNodo )
   {
      datos = datosNodo;
      nodoIzq = nodoDer = null; // el nodo no tiene hijos
   } // fin del constructor de NodoArbol

   // localiza el punto de inserción e inserta un nuevo nodo; ignora los valores duplicados
   public void insertar( int valorInsertar )
   {
      // inserta en el subárbol izquierdo
      if ( valorInsertar < datos )
      {
         // inserta nuevo NodoArbol
         if ( nodoIzq == null )
            nodoIzq = new NodoArbol( valorInsertar );
         else // continúa recorriendo el subárbol izquierdo
            nodoIzq.insertar( valorInsertar );
      } // fin de if
```

Figura 17.17 | Declaraciones de las clases `NodoArbol` y `Arbol` para un árbol de búsqueda binaria. (Parte 1 de 3).

```
         else if ( valorInsertar > datos ) // inserta en el subárbol derecho
         {
            // inserta nuevo NodoArbol
            if ( nodoDer == null )
               nodoDer = new NodoArbol( valorInsertar );
            else // continúa recorriendo el subárbol derecho
               nodoDer.insertar( valorInsertar );
         } // fin de else if
      } // fin del método insertar
} // fin de la clase NodoArbol

// definición de la clase Arbol
public class Arbol
{
   private NodoArbol raiz;

   // el constructor inicializa un Arbol vacío de enteros
   public Arbol()
   {
      raiz = null;
   } // fin del constructor de Arbol sin argumentos

   // inserta un nuevo nodo en el árbol de búsqueda binaria
   public void insertarNodo( int valorInsertar )
   {
      if ( raiz == null )
         raiz = new NodoArbol( valorInsertar ); // crea el nodo raíz aquí
      else
         raiz.insertar( valorInsertar ); // llama al método insertar
   } // fin del método insertarNodo

   // comienza el recorrido preorden
   public void recorridoPreorden()
   {
      ayudantePreorden( raiz );
   } // fin del método recorridoPreorden

   // método recursivo para realizar el recorrido preorden
   private void ayudantePreorden( NodoArbol nodo )
   {
      if ( nodo == null )
         return;

      System.out.printf( "%d ", nodo.datos ); // imprime los datos del nodo
      ayudantePreorden( nodo.nodoIzq );       // recorre el subárbol izquierdo
      ayudantePreorden( nodo.nodoDer );       // recorre el subárbol derecho
   } // fin del método ayudantePreorden

   // comienza recorrido inorden
   public void recorridoInorden()
   {
      ayudanteInorden( raiz );
   } // fin del método recorridoInorden

   // método recursivo para realizar el recorrido inorden
   private void ayudanteInorden( NodoArbol nodo )
   {
      if ( nodo == null )
         return;
```

Figura 17.17 | Declaraciones de las clases `NodoArbol` y `Arbol` para un árbol de búsqueda binaria. (Parte 2 de 3).

```
91
92          ayudanteInorden( nodo.nodoIzq );        // recorre el subárbol izquierdo
93          System.out.printf( "%d ", nodo.datos ); // imprime los datos del nodo
94          ayudanteInorden( nodo.nodoDer );        // recorre el subárbol derecho
95       } // fin del método ayudanteInorden
96
97       // comienza recorrido postorden
98       public void recorridoPostorden()
99       {
100         ayudantePostorden( raiz );
101      } // fin del método recorridoPostorden
102
103      // método recursivo para realizar el recorrido postorden
104      private void ayudantePostorden( NodoArbol nodo )
105      {
106         if ( nodo == null )
107            return;
108
109         ayudantePostorden( nodo.nodoIzq );       // recorre el subárbol izquierdo
110         ayudantePostorden( nodo.nodoDer );       // recorre el subárbol derecho
111         System.out.printf( "%d ", nodo.datos );  // imprime los datos del nodo
112      } // fin del método ayudantePostorden
113   } // fin de la clase Arbol
```

Figura 17.17 | Declaraciones de las clases `NodoArbol` y `Arbol` para un árbol de búsqueda binaria. (Parte 3 de 3).

Analicemos el programa del árbol binario. El método `main` de la clase `PruebaArbol` (figura 17.18) empieza creando una instancia de un objeto `Arbol` vacío y asigna su referencia a la variable `arbol` (línea 10). En las líneas 17 a 22 se generan 10 enteros al azar, cada uno de los cuales se inserta en el árbol binario mediante una llamada al método `insertarNodo` (línea 21). Después el programa realiza recorridos preorden, inorden y postorden (los cuales explicaremos en breve) de `arbol` (líneas 25, 28 y 31, respectivamente).

La clase `Arbol` (figura 17.17, líneas 44 a 113) tiene un campo `private` llamado `raiz` (línea 46); una referencia tipo `NodoArbol` al nodo raíz del árbol. El constructor de `Arbol` (líneas 49 a 52) inicializa `raiz` con `null` para indicar que el árbol está vacío. La clase contiene el método `insertarNodo` (líneas 55 a 61) para insertar un nuevo nodo en el árbol, además de los métodos `recorridoPreorden` (líneas 64 a 67), `recorridoInorden` (líneas 81 a 84) y `recorridoPostorden` (líneas 98 a 101) para empezar recorridos del árbol. Cada uno de estos métodos llama a un método utilitario recursivo para realizar las operaciones de recorrido en la representación interna del árbol. (En el capítulo 15 hablamos sobre la recursividad).

```
1   // Fig. 17.18: PruebaArbol.java
2   // Este programa prueba la clase Arbol.
3   import java.util.Random;
4   import com.deitel.jhtp7.cap17.Arbol;
5
6   public class PruebaArbol
7   {
8      public static void main( String args[] )
9      {
10        Arbol arbol = new Arbol();
11        int valor;
12        Random numeroAleatorio = new Random();
13
14        System.out.println( "Insertando los siguientes valores: " );
15
16        // inserta 10 enteros aleatorios de 0 a 99 en arbol
```

Figura 17.18 | Programa de prueba de un árbol binario. (Parte 1 de 2).

```
        for ( int i = 1; i <= 10; i++ )
        {
           valor = numeroAleatorio.nextInt( 100 );
           System.out.print( valor + " " );
           arbol.insertarNodo( valor );
        } // fin de for

        System.out.println ( "\n\nRecorrido preorden" );
        arbol.recorridoPreorden(); // realiza recorrido preorden de arbol

        System.out.println ( "\n\nRecorrido inorden" );
        arbol.recorridoInorden(); // realiza recorrido inorden de arbol

        System.out.println ( "\n\nRecorrido postorden" );
        arbol.recorridoPostorden(); // realiza recorrido postorden de arbol
        System.out.println();
     } // fin de main
  } // fin de la clase PruebaArbol
```

```
Insertando los siguientes valores:
17 54 3 30 95 69 85 88 16 30

Recorrido preorden
17 3 16 54 30 95 69 85 88

Recorrido inorden
3 16 17 30 54 69 85 88 95

Recorrido postorden
16 3 30 88 85 69 95 54 17
```

Figura 17.18 | Programa de prueba de un árbol binario. (Parte 2 de 2).

El método `insertarNodo` de la clase `Arbol` (líneas 55 a 61) determina primero si el árbol está vacío. De ser así, en la línea 58 se asigna un nuevo objeto `NodoArbol`, se inicializa el nodo con el entero que se insertará en el árbol y se asigna el nuevo nodo a la referencia `raiz`. Si el árbol no está vacío, en la línea 60 se hace una llamada al método `insertar` de `NodoArbol` (líneas 21 a 41). Este método utiliza la recursividad para determinar la posición del nuevo nodo en el árbol e inserta el nodo en esa posición. En un árbol de búsqueda binaria, un nodo puede insertarse solamente como nodo hoja.

El método `insertar` de `NodoArbol` compara el valor a insertar con el valor de datos en el nodo raíz. Si el valor a insertar es menor que los datos del nodo raíz (línea 24), el programa determina si el subárbol izquierdo está vacío (línea 27). De ser así, en la línea 28 se asigna un nuevo objeto `NodoArbol`, se inicializa con el entero que se insertará y se asigna el nuevo nodo a la referencia `nodoIzquierdo`. En caso contrario, en la línea 30 se hace una llamada recursiva a `insertar` para que se inserte el valor en el subárbol izquierdo. Si el valor a insertar es mayor que los datos del nodo raíz (línea 32), el programa determina si el subárbol derecho está vacío (línea 35). De ser así, en la línea 36 se asigna un nuevo objeto `NodoArbol`, se inicializa con el entero que se insertará y se asigna el nuevo nodo a la referencia `nodoDerecho`. En caso contrario, en la línea 38 se hace una llamada recursiva a `insertar` para que se inserte el valor en el subárbol derecho. Si el `valorInsertar` ya se encuentra en el árbol, simplemente se ignora.

Los métodos `recorridoInorden`, `recorridoPreorden` y `recorridoPostorden` llaman a los métodos ayudantes de `Arbol` llamados `ayudanteEnorden` (líneas 87 a 95), `ayudantePreorden` (líneas 70 a 78) y `ayudantePostorden` (líneas 104 a 112), respectivamente, para recorrer el árbol e imprimir los valores de los nodos. Los métodos ayudantes en la clase `Arbol` permiten al programador iniciar un recorrido sin tener que pasar el nodo `raiz` al método. La referencia `raiz` es un detalle de implementación que no debe ser accesible para el programador. Los métodos `recorridoInorden`, `recorridoPreorden` y `recorridoPostorden` simplemente toman la referencia privada `raiz` y la pasan al método ayudante apropiado para iniciar un recorrido del árbol. El caso base para cada método ayudante determina si la referencia que recibe es `null` y, de ser así, regresa inmediatamente.

El método `ayudanteInorden` (líneas 87 a 95) define los pasos para un recorrido inorden:

1. Recorrer el subárbol izquierdo con una llamada a `ayudanteInorden` (línea 92).
2. Procesar el valor en el nodo (línea 93).
3. Recorrer el subárbol derecho con una llamada a `ayudanteInorden` (línea 94).

El recorrido inorden no procesa el valor en un nodo sino hasta que se procesan los valores en el subárbol izquierdo de ese nodo. El recorrido inorden del árbol de la figura 17.19 es:

```
6 13 17 27 33 42 48
```

Observe que el recorrido inorden de un árbol de búsqueda binaria imprime los valores de los nodos en orden ascendente. El proceso de crear un árbol de búsqueda binaria ordena los datos de antemano; por lo tanto, a este proceso se le conoce como **ordenamiento de árbol binario**.

El método `ayudantePreorden` (líneas 70 a 78) define los pasos para un recorrido preorden:

1. Procesar el valor en el nodo (línea 75).
2. Recorrer el subárbol izquierdo con una llamada a `ayudantePreorden` (línea 76).
3. Recorrer el subárbol derecho con una llamada a `ayudantePreorden` (línea 77).

El recorrido preorden procesa el valor en cada uno de los nodos, a medida que se van visitando. Después de procesar el valor en un nodo dado, el recorrido preorden procesa los valores en el subárbol izquierdo y después los valores en el subárbol derecho. El recorrido preorden del árbol de la figura 17.19 es:

```
27 13 6 17 42 33 48
```

El método `ayudantePostorden` (líneas 104 a 112) define los pasos para un recorrido postorden:

1. Recorrer el subárbol izquierdo con una llamada a `ayudantePostorden` (línea 109).
2. Recorrer el subárbol derecho con una llamada a `ayudantePostorden` (línea 110).
3. Procesar el valor en el nodo (línea 111).

El recorrido postorden procesa el valor en cada nodo después de procesar los valores de todos los hijos de ese nodo. El `recorridoPostorden` del árbol de la figura 17.19 es:

```
6 17 13 33 48 42 27
```

El árbol de búsqueda binaria facilita la **eliminación de valores duplicados**. Al crear un árbol, la operación de inserción reconoce los intentos de insertar un valor duplicado, ya que éste sigue las mismas decisiones de “ir a la izquierda” o “ir a la derecha” en cada comparación, al igual que el valor original. Por lo tanto, la operación de inserción eventualmente comparará el valor duplicado con un nodo que contenga el mismo valor. En este punto, la operación de inserción puede decidir descartar el valor duplicado (como lo hicimos en este ejemplo).

Buscar en un árbol binario un valor que concuerde con una clave es un proceso rápido, especialmente para los árboles **estrechamente empaquetados** (o **balanceados**). En un árbol estrechamente empaquetado, cada nivel contiene aproximadamente el doble de elementos que el nivel anterior. La figura 17.19 es un árbol binario estrechamente empaquetado. Un árbol de búsqueda binaria estrechamente empaquetado con n elementos tiene $\log_2 n$ niveles. Por lo tanto, se requieren cuando mucho $\log_2 n$ comparaciones para encontrar una concordancia o determinar que no existe una. La búsqueda en un árbol de búsqueda binaria de 1000 elementos (estrechamente empaquetado) requiere cuando mucho de 10 comparaciones, ya que $2^{10} > 1000$. La búsqueda en un árbol de

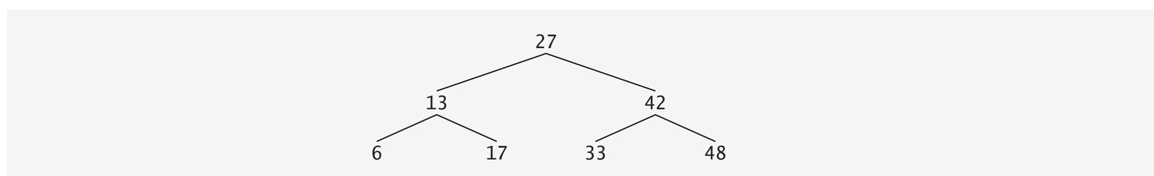


Figura 17.19 | Árbol de búsqueda binaria con siete valores.

búsqueda binaria de 1,000,000 de elementos (estrechamente empaquetado) requiere cuando mucho de 20 comparaciones, ya que $2^{20} > 1,000,000$.

Los ejercicios de este capítulo presentan algoritmos para varias operaciones más de árboles binarios, como eliminar un elemento de un árbol binario, imprimir un árbol binario en formato de árbol bidimensional y realizar un **recorrido en orden de niveles de un árbol binario**. En el recorrido en orden de niveles de un árbol binario se visitan sus nodos fila por fila, empezando en el nivel del nodo raíz. En cada nivel del árbol, un recorrido en orden de niveles visita los nodos de izquierda a derecha. Otros ejercicios de árboles binarios incluyen el permitir que un árbol de búsqueda binaria contenga valores duplicados, insertar valores de cadena en un árbol binario y determinar cuántos niveles hay en un árbol binario. En el capítulo 19 continuaremos con nuestra discusión sobre las estructuras de datos, al presentar las estructuras de datos incluidas en la API de Java.

17.10 Conclusión

En este capítulo aprendió acerca de las clases de envoltura de tipos, la conversión boxing y las estructuras dinámicas de datos, que aumentan y reducen su tamaño en tiempo de ejecución. Aprendió que cada tipo primitivo tiene su correspondiente clase de envoltura de tipo en el paquete `java.lang`. También vio que Java puede realizar conversiones entre valores primitivos y objetos de las clases de envoltura de tipos, mediante la conversión boxing.

Aprendió que las listas enlazadas son elementos de datos que están “alineados en una fila”. También vio que una aplicación puede realizar operaciones de inserción y eliminación de datos en cualquier parte de una lista enlazada. Aprendió que las estructuras de datos tipo pila y cola son versiones restringidas de listas. En cuanto a las pilas, vio que las operaciones de insertar y eliminar datos se realizan sólo en la parte superior. En cuanto a las colas que representan líneas de espera, vio que las inserciones se realizan en la parte final (cola) y las eliminaciones se realizan en la parte frontal (cabeza). También aprendió acerca de la estructura de datos tipo árbol binario. Vio un árbol de búsqueda binaria que facilita la búsqueda y el ordenamiento de los datos de alta velocidad, además de que se pueden eliminar los elementos de datos duplicados de una manera eficiente. A lo largo de este capítulo, aprendió a crear y empaquetar estas estructuras de datos para reutilizarlas y darles mantenimiento.

En el capítulo 18, Genéricos, presentaremos un mecanismo para declarar clases y métodos sin información específica sobre los tipos, de manera que las clases y métodos se puedan utilizar con muchos tipos distintos. Los genéricos se utilizan ampliamente en el conjunto integrado de estructuras de datos de Java, el cual se conoce como la API Colecciones, que veremos en el capítulo 19.

Resumen

Sección 17.1 Introducción

- Las estructuras de datos dinámicas pueden crecer y reducirse en tiempo de ejecución.
- Las listas enlazadas son colecciones de elementos de datos “alineados en una fila”; pueden insertarse y eliminarse elementos en cualquier parte de una lista enlazada.
- Las pilas son importantes en los compiladores y sistemas operativos; pueden insertarse y eliminarse elementos solamente en un extremo de una pila: su parte superior.
- En una cola se insertan elementos en la parte final (cola) y se eliminan de su parte inicial (cabeza).
- Los árboles binarios facilitan la búsqueda y ordenamiento de los datos de alta velocidad, la eliminación eficiente de elementos de datos duplicados, la representación de directorios del sistema de archivos y la compilación de expresiones en lenguaje máquina.

Sección 17.2 Clases de envoltura de tipos para los tipos primitivos

- Las clases de envoltura de tipos (por ejemplo, `Integer`, `Double`, `Boolean`) permiten a los programadores manipular valores de tipos primitivos como objetos. Los objetos de estas clases se pueden utilizar en colecciones y estructuras de datos que sólo pueden almacenar referencias a objetos, y no valores de tipos primitivos.

Sección 17.3 Autoboxing y autounboxing

- Una conversión boxing convierte un valor de un tipo primitivo en un objeto de su clase de envoltura de tipo correspondiente. Una conversión unboxing convierte un objeto de una clase de envoltura de tipo en un valor del tipo primitivo correspondiente.

- Java realiza conversiones boxing y unboxing de manera automática (a lo cual se le conoce como autoboxing y autounboxing).

Sección 17.4 Clases autorreferenciadas
- Una clase autorreferenciada contiene una referencia a otro objeto del mismo tipo de clase. Los objetos autorreferenciados pueden enlazarse entre sí para formar estructuras de datos dinámicas.

Sección 17.5 Asignación dinámica de memoria
- El límite para la asignación dinámica de memoria puede ser tan grande como la cantidad de memoria física disponible en la computadora, o la cantidad de espacio en disco disponible en un sistema con memoria virtual. A menudo, los límites son mucho más pequeños ya que la memoria disponible de la computadora debe compartirse entre muchos usuarios.
- Si no hay memoria disponible, se lanza una excepción `OutOfMemoryError`.

Sección 17.6 Listas enlazadas
- Una lista enlazada se utiliza mediante una referencia al primer nodo de la lista. Cada nodo subsiguiente se utiliza a través del miembro de referencia de enlace almacenado en el nodo anterior.
- Por convención, la referencia de enlace en el último nodo de una lista se establece en `null` para indicar el final de la lista.
- Un nodo puede contener datos de cualquier tipo, incluyendo objetos de otras clases.
- Una lista enlazada es apropiada cuando el número de elementos de datos que se van a almacenar es impredecible. Las listas enlazadas son dinámicas, por lo que su longitud puede incrementarse o reducirse, según sea necesario.
- El tamaño de un arreglo "convencional" en Java no puede alterarse; se fija al momento de su creación.
- Las listas enlazadas pueden mantenerse en orden con sólo insertar cada nuevo elemento en el punto apropiado de la lista.
- Generalmente, los nodos de las listas no se almacenan contiguamente en memoria. En vez de ello, son adyacentes en forma lógica.

Sección 17.7 Pilas
- A una pila se le conoce como estructura de datos UEPS (último en entrar, primero en salir). Los métodos principales usados para manipular una pila son empujar (`push`) y sacar (`pop`). El método `push` agrega un nuevo nodo a la parte superior de la pila. El método `pop` elimina un nodo de la parte superior de la pila y devuelve el objeto datos del nodo que se quitó.
- Las pilas tienen muchas aplicaciones interesantes. Cuando se hace la llamada a un método, el método llamado debe saber cómo regresar a su invocador, por lo que la dirección de retorno se mete en la pila de ejecución del programa. Si ocurre una serie de llamadas a métodos, los valores de retorno sucesivos se meten en la pila en el orden "último en entrar, primero en salir", para que cada método pueda regresar a su invocador. La pila de ejecución del programa contiene el espacio creado para las variables locales en cada invocación de un método. Cuando el método regresa a su invocador, el espacio para las variables locales de ese método se saca de la pila y esas variables dejan de ser conocidas por el programa.
- Los compiladores utilizan pilas para evaluar expresiones aritméticas y generar código en lenguaje máquina para procesar las expresiones.
- La técnica de implementar cada método de la pila como una llamada a un método de `Lista` se conoce como delegación; el método invocado de la pila delega la llamada al método apropiado de `Lista`.

Sección 17.8 Colas
- Una cola es similar a la línea para pagar en un supermercado: la primera persona en la línea es atendida primero, y los demás clientes entran a la línea sólo por su parte final, esperando a ser atendidos.
- Los nodos de una cola se eliminan sólo desde el principio de la misma y se insertan sólo al final de ésta. Por esta razón, a una cola se le conoce como estructura de datos PEPS (primero en entrar, primero en salir).
- Las operaciones para insertar y eliminar en una cola se conocen como `enqueue` (agregar a la cola) y `dequeue` (retirar de la cola).
- Las colas tienen muchos usos en los sistemas computacionales. La mayoría de las computadoras tienen sólo un procesador, por lo que sólo pueden atender a una aplicación a la vez. Las entradas para las otras aplicaciones se colocan en una cola. La entrada al frente de la cola es la siguiente que recibe atención. Cada entrada avanza gradualmente al frente de la cola, a medida que los usuarios reciben atención.

Sección 17.9 Árboles
- Un árbol es una estructura de datos bidimensional, no lineal. Los nodos de un árbol contienen dos o más enlaces.

- Un árbol binario es un árbol cuyos nodos contienen dos enlaces. El nodo raíz es el primer nodo en un árbol.
- Cada enlace en el nodo raíz hace referencia a un hijo. El hijo izquierdo es el primer nodo en el subárbol izquierdo, y el hijo derecho es el primer nodo en el subárbol derecho.
- Los hijos de un nodo se llaman hermanos. Un nodo sin hijos se llama nodo hoja.
- En un árbol de búsqueda binaria sin valores de nodo duplicados, los valores en cualquier subárbol izquierdo son menores que el valor en su nodo padre, y los valores en cualquier subárbol derecho son mayores que el valor en su nodo padre. En un árbol de búsqueda binaria, un nodo puede insertarse solamente como nodo hoja.
- El recorrido inorden de un árbol de búsqueda binaria procesa los valores de los nodos en orden ascendente.
- En un recorrido preorden, el valor en cada uno de los nodos se procesa a medida que se van visitando. Después se procesan los valores en el subárbol izquierdo y, por último, los valores en el subárbol derecho.
- En un recorrido postorden, el valor en cada uno de los nodos se procesa después de los valores de sus hijos.
- El árbol de búsqueda binaria facilita la eliminación de valores duplicados. Al crear un árbol se reconocen los intentos de insertar un valor duplicado, ya que éste sigue las mismas decisiones de "ir a la izquierda" o "ir a la derecha" en cada comparación, al igual que el valor original. Por lo tanto, eventualmente se compara el valor duplicado con un nodo que contenga el mismo valor. El valor duplicado puede descartarse en este punto.
- Buscar en un árbol binario un valor que concuerde con una clave es también un proceso rápido, especialmente para los árboles estrechamente empaquetados. En un árbol estrechamente empaquetado, cada nivel contiene aproximadamente el doble de elementos que el nivel anterior. Por lo tanto, un árbol de búsqueda binaria estrechamente empaquetado con n elementos tiene $\log_2 n$ niveles, por lo que tendrían que hacerse cuando mucho $\log_2 n$ comparaciones para encontrar una concordancia o determinar que no existe una. La búsqueda en un árbol de búsqueda binaria de 1000 elementos (estrechamente empaquetado) requiere cuando mucho de 10 comparaciones, ya que $2^{10} > 1000$. La búsqueda en un árbol de búsqueda binaria de 1,000,000 de elementos (estrechamente empaquetado) requiere cuando mucho de 20 comparaciones, ya que $2^{20} > 1,000,000$.

Terminología

algoritmos recursivos para recorrer árboles
árbol
árbol balanceado
árbol binario
árbol de búsqueda binaria
árbol empaquetado
autoboxing
autounboxing
`Boolean`, clase
`Byte`, clase
clase autorreferenciada
clases de envoltura de tipos
cola
conversión boxing
conversión unboxing
`Character`, clase
delegar la llamada a un método
dequeue
eliminación de valores duplicados
eliminar un nodo
enqueue
estructura de datos lineal
estructura de datos no lineal
estructura dinámica de datos
`Float`, clase
hijo derecho
hijo izquierdo
hijos de un nodo
insertar un nodo
`Integer`, clase
lista enlazada
`Long`, clase
método predicado
nodo
nodo hijo
nodo hoja
nodo padre
nodo raíz
`null`, referencia
ordenamiento de árboles binarios
`OutOfMemoryError`
parte final de una cola
parte inicial (cabeza) de una cola
parte superior de una pila
PEPS (primero en entrar, primero en salir)
pila
pila de ejecución del programa
pop
push
recorrido
recorrido en orden de niveles de un árbol binario
recorrido inorden de un árbol binario
recorrido postorden de un árbol binario
recorrido preorden de un árbol binario
`Short`, clase
subárbol
subárbol derecho
subárbol izquierdo
UEPS (último en entrar, primero en salir)
visitar un nodo

Ejercicios de autoevaluación

17.1 Llene los espacios en blanco en cada uno de los siguientes enunciados:

a) Una clase ____________________ se utiliza para formar estructuras de datos dinámicas que pueden crecer y reducirse en tiempo de ejecución.
b) Una ____________________ es una versión restringida de una lista enlazada, en la que pueden insertarse y eliminarse nodos solamente desde el principio de la lista.
c) Un método que no altera una lista enlazada, sino que sólo la analiza para determinar si está vacía, se conoce como método ____________________.
d) A una cola se le conoce como estructura de datos ____________________, ya que los primeros nodos que se insertan son los primeros que se eliminan.
e) La referencia al siguiente nodo en una lista enlazada se conoce como un ____________________.
f) Al proceso de reclamar automáticamente la memoria asignada en forma dinámica se le conoce como ____________________.
g) Una ____________________ es una versión restringida de una lista enlazada, en la que pueden insertarse nodos al final de la lista y eliminarse solamente desde el principio.
h) Un ____________________ es una estructura de datos bidimensional no lineal, que contiene nodos con dos o más enlaces.
i) A una pila se le conoce como estructura de datos ____________________, ya que el último nodo insertado es el primero que se elimina.
j) Los nodos de un árbol ____________________ contienen dos miembros de enlace.
k) El primer nodo de un árbol es el nodo ____________________.
l) Cada enlace en el nodo de un árbol hace referencia a un ____________________ o ____________________ de ese nodo.
m) El nodo de un árbol que no tiene hijos se llama nodo ____________________.
n) Los tres algoritmos de recorrido que mencionamos en el texto para los árboles de búsqueda binaria son ____________________, ____________________ y ____________________.
o) Suponiendo que `miArreglo` contiene referencias a objetos `Double`, una ____________________ ocurre cuando se ejecuta la instrucción "`double numero = miArreglo[ 0 ];`".
p) Suponiendo que `miArreglo` contiene referencias a objetos `Double`, una ____________________ ocurre cuando se ejecuta la instrucción "`miArreglo[ 0 ] = 1.25;`".

17.2 ¿Cuáles son las diferencias entre una lista enlazada y una pila?

17.3 ¿Cuáles son las diferencias entre una pila y una cola?

17.4 Tal vez un título más apropiado para este capítulo hubiera sido "Estructuras de datos reutilizables". Escriba sus comentarios acerca de cómo contribuyen cada una de las siguientes entidades o conceptos a la reutilización de las estructuras de datos:

a) clases
b) herencia
c) composición

17.5 Proporcione manualmente los recorridos inorden, preorden y postorden del árbol de búsqueda binaria de la figura 17.20.

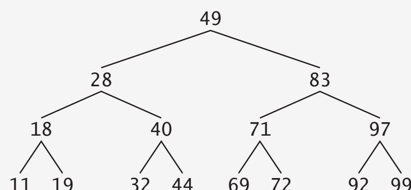


Figura 17.20 | Árbol de búsqueda binaria con 15 nodos.

Respuestas a los ejercicios de autoevaluación

17.1 a) autorreferenciada. b) pila. c) predicado. d) PEPS (primero en entrar, primero en salir). e) enlace. f) recolección de basura. g) cola. h) árbol. i) UEPS (último en entrar, primero en salir). j) binario. k) raíz. l) hijo o subárbol. m) hoja. n) inorden, preorden, postorden. o) conversión autounboxing. p) conversión autoboxing.

17.2 Es posible insertar y eliminar un nodo en cualquier lugar de una lista enlazada. Los nodos en una pila pueden insertarse solamente en la parte superior y eliminarse desde la parte superior de una pila.

17.3 Una cola es una estructura de datos PEPS que tiene referencias tanto a su parte inicial como a su parte final, de manera que pueden insertarse nodos al final y eliminarse del principio de la cola. Una pila es una estructura de datos UEPS que tiene una sola referencia a la parte superior de la pila, en donde se llevan a cabo las operaciones de inserción y eliminación de nodos.

17.4
a) Las clases nos permiten crear tantas instancias de todos los objetos de estructura de datos de cierto tipo (es decir, clase) como sea necesario.
b) La herencia permite a una subclase reutilizar la funcionalidad de una superclase. Los métodos public y protected de una superclase pueden utilizarse a través de una subclase, para eliminar la lógica duplicada.
c) La composición permite a una clase reutilizar código almacenando una referencia a una instancia de otra clase en un campo. Los métodos públicos de la clase miembro pueden ser llamados por los métodos de la clase que contiene la referencia.

17.5 El recorrido inorden es:

```
11 18 19 28 32 40 44 49 69 71 72 83 92 97 99
```

El recorrido preorden es:

```
49 28 18 11 19 40 32 44 83 71 69 72 97 92 99
```

El recorrido postorden es:

```
11 19 18 32 44 40 28 69 72 71 92 99 97 83 49
```

Ejercicios

17.6 Escriba un programa para concatenar dos objetos de lista enlazada de caracteres. La clase `ConcatenarLista` debe incluir un método llamado `concatenar` que tome referencias a ambos objetos lista como argumentos y que concatene la segunda lista con la primera.

17.7 Escriba un programa para fusionar dos objetos de lista ordenada de enteros en un solo objeto de lista ordenada de enteros. El método `fusionar` de la clase `FusionarLista` debe recibir referencias a cada uno de los objetos lista que se van a fusionar, y debe devolver una referencia al objeto lista fusionado.

17.8 Escriba un programa para insertar 25 enteros aleatorios de 0 a 100 en orden, en un objeto lista enlazada. El programa deberá calcular la suma de los elementos y el promedio de punto flotante de los elementos.

17.9 Escriba un programa para crear un objeto lista enlazada de 10 caracteres, y que luego cree un segundo objeto lista que contenga una copia de la primera lista, pero en orden inverso.

17.10 Escriba un programa que reciba una línea de texto como entrada y que utilice un objeto pila para imprimir las palabras de la línea en orden inverso.

17.11 Escriba un programa que utilice una pila para determinar si una cadena es un palíndromo (es decir, que la cadena se deletree en forma idéntica, tanto al revés como al derecho). El programa debe ignorar espacios y puntuación.

17.12 Los compiladores utilizan pilas para ayudar en el proceso de evaluar expresiones y generar código en lenguaje máquina. En este ejercicio y en el siguiente, investigaremos cómo los compiladores evalúan expresiones aritméticas que consisten solamente de constantes, operadores y paréntesis.

Los humanos generalmente escriben expresiones como `3 + 4` y `7 / 9`, en donde el operador (`+` o `/` aquí) se escribe entre sus operandos; a esta notación se le conoce como *notación infijo*. Las computadoras "prefieren" la *notación postfijo*, en donde el operador se escribe a la derecha de sus dos operandos. Las anteriores expresiones infijo aparecerían en notación postfijo como `3 4 +` y `7 9 /`, respectivamente.

Para evaluar una expresión infijo compleja, un compilador primero convertiría la expresión en notación postfijo y evaluaría la versión postfijo de la expresión. Cada uno de estos algoritmos requiere solamente de una pasada de izquierda

a derecha de la expresión. Cada algoritmo utiliza un objeto pila para dar soporte a su operación y, en cada algoritmo, la pila se utiliza para un propósito distinto.

En este ejercicio, usted escribirá una versión en Java del algoritmo de conversión infijo a postfijo. En el siguiente ejercicio, usted escribirá una versión en Java del algoritmo de evaluación de expresiones postfijo. En un ejercicio posterior, descubrirá que el código que escriba en este ejercicio podrá ayudarle a implementar un compilador completamente funcional.

Escriba la clase `ConvertidorInfijoAPostfijo` para convertir una expresión aritmética infijo ordinaria (suponga que se escribe una expresión válida) con enteros de un solo dígito, como:

```
(6 + 2) * 5 - 8 / 4
```

en una expresión postfijo. La versión postfijo de la expresión infijo anterior es (observe que no se necesitan paréntesis):

```
6 2 + 5 * 8 4 / -
```

El programa debe leer la expresión y colocarla en la variable `StringBuffer infijo`, y utilizar una de las clases de pila implementadas en este capítulo para ayudar a crear la expresión postfijo en la variable `StringBuffer postfijo`. El algoritmo para crear una expresión postfijo es el siguiente:

a) Meter un paréntesis izquierdo `'('` en la pila.
b) Anexar un paréntesis derecho `')'` al final de `infijo`.
c) Mientras que la pila no esté vacía, leer `infijo` de izquierda a derecha y hacer lo siguiente:
   - Si el carácter actual en `infijo` es un dígito, anexarlo a `postfijo`.
   - Si el carácter actual en `infijo` es un paréntesis izquierdo, meterlo a la pila.
   - Si el carácter actual en `infijo` es un operador:
     - Sacar los operadores (si los hay) de la parte superior de la pila, mientras tengan igual o mayor precedencia que el operador actual, y anexar los operadores que se sacaron a `postfijo`.
     - Meter en la pila el carácter actual de `infijo`.
   - Si el carácter actual en `infijo` es un paréntesis derecho:
     - Sacar operadores de la parte superior de la pila y anexarlos a `postfijo`, hasta que haya un paréntesis izquierdo en la parte superior de la pila.
     - Sacar (y descartar) el paréntesis izquierdo de la pila.

Las siguientes operaciones aritméticas se permiten en una expresión:

+ suma
— resta
* multiplicación
/ división
^ exponenciación
% residuo

La pila debe mantenerse con nodos de pila que contengan, cada uno, una variable de instancia y una referencia al siguiente nodo de la pila. Algunos de los métodos que puede proporcionar son:

a) El método `convertirAPostfijo`, que convierte la expresión infijo a notación postfijo.
b) El método `esOperador`, el cual determina si `c` es un operador.
c) El método `precedencia`, que determina si la precedencia de `operador1` (de la expresión infijo) es menor, igual o mayor que la precedencia de `operador2` (de la pila). El método devuelve `true` si `operador1` tiene menor precedencia que `operador2`. En caso contrario, se devuelve `false`.
d) El método `parteSuperiorPila` (éste debe agregarse a la clase pila), que devuelve el valor de la parte superior de la pila sin sacarlo de la misma.

17.13 Escriba la clase `EvaluadorPostfijo`, el cual evalúa una expresión postfijo como:

```
6 2 + 5 * 8 4 / -
```

El programa debe leer una expresión postfijo que consista de dígitos y operadores, para después colocarla en un objeto `StringBuffer`. Utilizando versiones modificadas de los métodos de pila implementados anteriormente en este capítulo, el programa deberá explorar la expresión y evaluarla (suponiendo que sea válida). El algoritmo es el siguiente:

a) Anexar un paréntesis derecho (`')'`) al final de la expresión postfijo. Cuando se encuentre el carácter de paréntesis derecho, ya no habrá nada más qué procesar.

b) Cuando no se encuentre el carácter de paréntesis derecho, leer la expresión de izquierda a derecha.
Si el carácter actual es un dígito, hacer lo siguiente:
Meter su valor entero en la pila (el valor entero de un carácter tipo dígito es su valor en el conjunto de caracteres de la computadora menos el valor de `'0'` en Unicode).
En caso contrario, si el carácter actual es un *operador*:
Sacar los dos elementos superiores de la pila y colocarlos en las variables `x` y `y`.
Calcular `y` *operador* `x`.
Meter el resultado del cálculo en la pila.
c) Cuando se encuentre el paréntesis derecho en la expresión, sacar el valor superior de la pila. Éste es el resultado de la expresión postfijo.

[*Nota:* en el inciso b) anterior (con base en la expresión de ejemplo al principio de este ejercicio), si el operador es `'/'`, el valor superior de la pila es `2` y el siguiente elemento en la pila es `8`, entonces sacar `2` y colocarlo en `x`, sacar `8` y colocarlo en `y`, evaluar `8 / 2` y meter el resultado (`4`) de vuelta en la pila. Esta nota también se aplica al operador `'-'`]. Las operaciones aritméticas permitidas en una expresión son:

+ suma
— resta
* multiplicación
/ división
^ exponenciación
% residuo

La pila debe mantenerse con una de las clases de pila que se presentaron en este capítulo. Tal vez usted pueda proporcionar los siguientes métodos:

a) El método `evaluarExpresionPostfijo`, el cual evalúa la expresión postfijo.
b) El método `calcular`, el cual evalúa la expresión `op1 operador op2`.
c) El método `push`, que mete un valor en la pila.
d) El método `pop`, que saca un valor de la pila.
e) El método `estaVacia`, que determina si la pila está vacía.
f) El método `imprimirPila`, el cual imprime la pila.

17.14 Modifique el programa evaluador de expresiones postfijo del ejercicio 17.13, de manera que pueda procesar operandos enteros mayores que 9.

17.15 *(Simulación de supermercado)* Escriba un programa que simule una línea para pagar en un supermercado. La línea es un objeto cola. Los clientes (es decir, los objetos cliente) llegan en intervalos enteros aleatorios de 1 a 4 minutos. Además, a cada cliente se le atiende en intervalos enteros aleatorios de 1 a 4 minutos. Obviamente, los ritmos necesitan balancearse. Si el ritmo promedio de llegadas es mayor que el ritmo promedio de atención, la cola crecerá infinitamente. Incluso con ritmos "balanceados", el factor aleatorio puede aún provocar largas líneas. Ejecute la simulación del supermercado durante un día de 12 horas (720 minutos), utilizando el siguiente algoritmo:

a) Elegir un entero aleatorio entre 1 y 4 para determinar el minuto en el que debe llegar el primer cliente.
b) Al momento en que llegue el cliente:
Determinar el tiempo de atención del cliente (entero aleatorio de 1 a 4).
Empezar a atender al cliente.
Programar la hora de llegada del siguiente cliente (se suma un entero aleatorio de 1 a 4 al tiempo actual).
c) Para cada minuto del día:
Si llega el siguiente cliente, hay que proceder de la siguiente manera:
Decirlo así.
Poner al cliente en la cola.
Programar la hora de llegada del siguiente cliente.
Si se terminó de atender al último cliente:
Decirlo así.
Sacar de la cola al siguiente cliente al que se va a atender.
Determinar el tiempo requerido para dar servicio al cliente (se suma un entero aleatorio del 1 al 4 al tiempo actual).

Ahora ejecute su simulación durante 720 minutos y responda a cada una de las siguientes preguntas:

a) ¿Cuál es el máximo número de clientes en la cola, en cualquier momento dado?

b) ¿Cuál es el tiempo de espera más largo que experimenta un cliente?
c) ¿Qué ocurre si el intervalo de llegada se cambia de 1 a 4 minutos por un intervalo de 1 a 3 minutos?

17.16 Modifique las figuras 17.17 y 17.18 para permitir que el árbol binario contenga valores duplicados.

17.17 Escriba un programa con base en el programa de las figuras 17.17 y 17.18, que reciba como entrada una línea de texto, divida la oración en palabras separadas (tal vez quiera utilizar la clase `StreamTokenizer` del paquete `java.io`), las inserte en un árbol de búsqueda binaria e imprima los recorridos inorden, preorden y postorden del árbol.

17.18 En este capítulo vimos que la eliminación de duplicados es un proceso bastante simple cuando se crea un árbol de búsqueda binaria. Describa cómo llevaría a cabo la eliminación de duplicados utilizando sólo un arreglo unidimensional. Compare el rendimiento de la eliminación de valores duplicados con base en arreglos y el rendimiento de la eliminación de duplicados con base en árboles de búsqueda binaria.

17.19 Escriba un método llamado `profundidad` que reciba un árbol binario y determine cuántos niveles tiene.

17.20 *(Imprimir una lista en forma recursiva y en forma inversa)* Escriba un método llamado `imprimirListaAlReves` que imprima en forma recursiva los elementos en un objeto lista enlazada, en orden inverso. Escriba un programa de prueba para crear una lista ordenada de enteros e imprimir la lista en orden inverso.

17.21 *(Buscar en una lista en forma recursiva)* Escriba un método llamado `buscarLista` que busque en forma recursiva en un objeto lista enlazada un valor específico. El método `buscarLista` deberá devolver una referencia al valor, si es que lo encuentra; en caso contrario, deberá devolver `null`. Use su método en un programa de prueba para crear una lista de enteros. El programa deberá pedir al usuario un valor a localizar en la lista.

17.22 *(Eliminación en árboles binarios)* En este ejercicio hablaremos sobre cómo eliminar elementos de los árboles de búsqueda binaria. El algoritmo de eliminación no es tan simple como el de inserción. Al eliminar un elemento puede haber tres casos: que el elemento esté contenido en un nodo hoja (es decir, que no tenga hijos), que esté contenido en un nodo que tenga un hijo, o que esté contenido en un nodo con dos hijos.

Si el elemento que se va a eliminar está contenido en un nodo hoja, este nodo se elimina y a la referencia en el nodo padre se le asigna el valor nulo.

Si el elemento que se eliminará está contenido en un nodo con un hijo, a la referencia en el nodo padre se le asigna el nodo hijo y se elimina el nodo que contenga el elemento de datos. Esto hace que el nodo hijo ocupe el lugar del nodo eliminado en el árbol.

El último caso es el más difícil. Cuando se elimina un nodo con dos hijos, otro nodo en el árbol debe tomar su lugar. Sin embargo, la referencia en el nodo padre no puede simplemente asignarse de manera que haga referencia a uno de los hijos del nodo que se va a eliminar. En la mayoría de los casos, el árbol de búsqueda binaria resultante no se adhiere a la siguiente característica de los árboles de búsqueda binaria (sin valores duplicados): *Los valores en cualquier subárbol izquierdo son menores que el valor en el nodo padre, y los valores en cualquier subárbol derecho son mayores que el valor en el nodo padre.*

¿Cuál nodo debe utilizarse como *nodo de reemplazo* para mantener esta característica? Debe ser el nodo que contenga el valor más grande en el árbol, pero que sea menor que el valor en el nodo que se eliminará, o el nodo que contenga el valor más pequeño en el árbol, pero que sea mayor que el valor en el nodo que se va a eliminar. Consideremos el nodo con el valor más pequeño. En un árbol de búsqueda binaria, el valor más grande que sea menor que el valor de un padre se encuentra en el subárbol izquierdo del nodo padre y se garantiza que estará contenido en el nodo que se encuentre más a la derecha del subárbol. Para localizar este nodo hay que avanzar por el subárbol izquierdo hacia abajo y a la derecha, hasta que la referencia al hijo derecho del nodo actual sea nula. Ahora estamos haciendo referencia al nodo de reemplazo, que es un nodo hoja o un nodo con un hijo a su izquierda. Si el nodo de reemplazo es un nodo hoja, los pasos para llevar a cabo la eliminación son los siguientes:

a) Almacenar la referencia al nodo que se eliminará en una variable de referencia temporal.
b) Hacer que la referencia en el padre del nodo que se va a eliminar haga referencia al nodo de reemplazo.
c) Asignar a la referencia en el padre del nodo de reemplazo el valor `null`.
d) Hacer que la referencia al subárbol derecho en el nodo de reemplazo haga referencia al subárbol derecho del nodo que se eliminará.
e) Hacer que la referencia al subárbol izquierdo en el nodo de reemplazo haga referencia al subárbol izquierdo del nodo que se va a eliminar.

Los pasos de eliminación para el caso de un nodo de reemplazo con un hijo izquierdo son similares a los pasos para un nodo de reemplazo sin hijos, sólo que el algoritmo debe también desplazar al hijo hacia la posición del nodo de reemplazo en el árbol. Si el nodo de reemplazo es un nodo con un hijo izquierdo, los pasos para llevar a cabo la eliminación son los siguientes:

a) Almacenar la referencia al nodo que se va a eliminar en una variable de referencia temporal.
b) Hacer que la referencia en el padre del nodo que se va a eliminar haga referencia al nodo de reemplazo.
c) Hacer que la referencia en el padre del nodo de reemplazo haga referencia al hijo izquierdo del nodo de reemplazo.
d) Hacer que la referencia al subárbol derecho en el nodo de reemplazo haga referencia al subárbol derecho del nodo que se va a eliminar.
e) Hacer que la referencia al subárbol izquierdo en el nodo de reemplazo haga referencia al subárbol izquierdo del nodo que se va a eliminar.

Escriba el método `eliminarNodo`, que debe tomar como argumento el valor a eliminar. El método `eliminarNodo` deberá localizar en el árbol el nodo que contenga el valor a eliminar, y deberá utilizar los algoritmos aquí descritos para eliminar el nodo. Si no se encuentra el valor en el árbol, el método deberá imprimir un mensaje que indique si el valor se eliminó. Modifique el programa de las figuras 17.17 y 17.18 para utilizar este método. Después de eliminar un elemento, llame a los métodos `recorridoInorden`, `recorridoPreorden` y `recorridoPostorden` para confirmar que la operación de eliminación se haya llevado a cabo correctamente.

17.23 *(Búsqueda en un árbol binario)* Escriba el método `busquedaArbolBinario`, para tratar de localizar un valor especificado en un objeto árbol de búsqueda binaria. El método deberá tomar como argumento una clave de búsqueda a localizar. Si se encuentra el nodo que contenga la clave de búsqueda, el método deberá devolver una referencia a ese nodo; en caso contrario, el método deberá devolver una referencia nula.

17.24 *(Recorrido de un árbol binario en orden de niveles)* El programa de las figuras 17.17 y 17.18 demostró el uso de tres métodos recursivos para recorrer un árbol binario: los recorridos inorden, preorden y postorden. En este ejercicio presentamos el *recorrido en orden de niveles* de un árbol binario, en el cual los valores de los nodos se imprimen nivel por nivel, empezando en el nivel del nodo raíz. Los nodos en cada nivel se imprimen de izquierda a derecha. El recorrido en orden de niveles no es un algoritmo recursivo. Utiliza un objeto cola para controlar la impresión en pantalla de los nodos. El algoritmo es el siguiente:

a) Insertar el nodo raíz en la cola.
b) Mientras haya nodos restantes en la cola:
    Obtener el siguiente nodo en la cola.
    Imprimir el valor del nodo.
    Si la referencia al hijo izquierdo del nodo no es nula:
        Insertar el nodo del hijo izquierdo en la cola.
    Si la referencia al hijo derecho del nodo no es nula:
        Insertar el nodo del hijo derecho en la cola.

Escriba el método `ordenNiveles` para llevar a cabo un recorrido en orden de niveles de un objeto árbol binario. Modifique el programa de las figuras 17.17 y 17.18 para utilizar este método. (*Nota:* también necesitará utilizar los métodos de procesamiento de colas de la figura 17.13 en este programa).

17.25 *(Imprimir árboles)* Escriba un método recursivo llamado `mostrarArbol` para mostrar un objeto árbol binario en la pantalla. El método deberá mostrar el árbol fila por fila, con la parte superior del mismo a la izquierda de la pantalla y la parte inferior hacia la derecha de la pantalla. Cada fila se debe mostrar en forma vertical. Por ejemplo, el árbol binario que aparece en la figura 17.20 debe mostrarse en pantalla como se indica en la figura 17.21.

El nodo hoja que se encuentra más a la derecha en el árbol aparece en la parte superior de la pantalla, en la columna que está más a la derecha, y el nodo raíz aparece a la izquierda de la pantalla. Cada columna de salida empieza cinco espacios a la derecha de la columna anterior. El método `mostrarArbol` debe recibir un argumento llamado `totalEspacios`, el cual representa el número de espacios que anteceden al valor que va a mostrarse en pantalla. (Esta variable debe empezar en cero, para que el nodo raíz se muestre a la izquierda de la pantalla). El método utiliza un recorrido inorden modificado para mostrar el árbol en pantalla; empieza en el nodo que está más a la derecha en el árbol y avanza en retroceso hacia la izquierda. El algoritmo es el siguiente:

Mientras la referencia al nodo actual no sea nula:
    Llamar en forma recursiva a `mostrarArbol` con el subárbol derecho del nodo actual y `totalEspacios + 5`.
    Utilizar una instrucción `for` para contar de 1 a `totalEspacios` e imprimir espacios.
    Mostrar el valor en el nodo actual.
    Hacer que la referencia al nodo actual haga referencia al subárbol izquierdo del nodo actual.
    Incrementar `totalEspacios` en 5.

```
                99
          97
                92
      83
                72
          71
                69
49
                44
          40
                32
      28
                19
          18
                11
```

Figura 17.21 | Resultados de ejemplo del método recursivo `mostrarArbol`.

Sección especial: construya su propio compilador

En los ejercicios 7.34 y 7.35 presentamos el Lenguaje Máquina Simpletron (LMS) y usted implementó un simulador de computadora Simpletron para ejecutar programas escritos en LMS. En esta sección crearemos un compilador que convierta los programas escritos en un lenguaje de programación de alto nivel a LMS. Esta sección "enlaza" entre sí todo el proceso de programación. Usted escribirá programas en este nuevo lenguaje de alto nivel, los compilará en el compilador que va a construir y los ejecutará en el simulador que construyó en el ejercicio 7.35. Usted deberá hacer todo el esfuerzo posible por implementar su compilador con un enfoque orientado a objetos.

17.26 *(El lenguaje Simple)* Antes de empezar a construir el compilador, hablaremos sobre un lenguaje de alto nivel simple pero poderoso, similar a las primeras versiones del popular lenguaje BASIC. Llamaremos a este lenguaje *Simple*. Cada *instrucción* de Simple consiste de un *número de línea* y de una *instrucción* de Simple. Los números de línea deben aparecer en orden ascendente. Cada instrucción empieza con uno de los siguientes *comandos* de Simple: `rem`, `input`, `let`, `print`, `goto`, `if/goto` o `end` (vea la figura 17.22). Todos los comandos excepto `end` pueden utilizarse en forma repetida. Simple evalúa solamente las expresiones de enteros que utilizan los operadores `+`, `-`, `*` y `/`. Estos operadores tienen la misma precedencia que en Java. Pueden utilizarse paréntesis para cambiar el orden de evaluación de una expresión.

Nuestro compilador de Simple reconoce solamente letras en minúscula; por lo tanto, todos los caracteres en un archivo de Simple deben estar en minúsculas. (Las letras mayúsculas producen un error de sintaxis a menos que aparezcan en una instrucción `rem`, en cuyo caso se ignoran). Un *nombre de variable* es una sola letra. Simple no permite el uso de nombres descriptivos para las variables, por lo que éstas deben explicarse en comentarios para indicar su uso en un programa. Simple utiliza solamente variables enteras. Simple no tiene declaraciones de variables; con sólo mencionar el nombre de una variable en un programa, ésta se declara y se inicializa con cero. La sintaxis de Simple no permite la manipulación de cadenas (leer una cadena, escribir una cadena, comparar cadenas, etcétera). Si se encuentra una cadena en un programa de Simple (después de un comando distinto de `rem`), el compilador genera un error de sintaxis. La primera versión de nuestro compilador supone que los programas de Simple se introducen correctamente. En el ejercicio 17.29 pedimos al lector que modifique el compilador para llevar a cabo la comprobación de errores de sintaxis.

Simple utiliza la instrucción `if/goto` condicional y la instrucción `goto` incondicional para alterar el flujo de control durante la ejecución del programa. Si la condición en la instrucción `if/goto` es verdadera, el control se transfiere a una línea específica del programa. Los siguientes operadores relacionales y de igualdad son válidos en una instrucción `if/goto`: `<`, `>`, `<=`, `>=`, `==` o `!=`. La precedencia de estos operadores es la misma que en Java.

Consideremos ahora varios programas para demostrar las características de Simple. El primer programa (figura 17.23) lee dos enteros del teclado, almacena los valores en las variables `a` y `b`, calcula e imprime su suma (almacenada en la variable `c`).

El programa de la figura 17.24 determina e imprime el mayor de dos enteros. Los enteros se introducen desde el teclado y se almacenan en `s` y `t`. La instrucción `if/goto` evalúa la condición `s >= t`. Si es verdadera, el control se transfiere a la línea `90` y se muestra el valor de `s` en pantalla; en caso contrario se muestra `t` y el control se transfiere a la instrucción `end` de la línea `99`, en donde termina el programa.

Comando	Instrucción de ejemplo	Descripción
`rem`	`50 rem este es un comentario`	Cualquier texto después del comando `rem` es para fines de documentación solamente, por lo que el compilador lo ignora.
`input`	`30 input x`	Mostrar un signo de interrogación para pedir al usuario que introduzca un entero. Leer ese entero desde el teclado y almacenarlo en `x`.
`let`	`80 let u = 4 * (j - 56)`	Asignar a `u` el valor de `4 * (j - 56)`. Observe que puede aparecer una expresión arbitrariamente compleja a la derecha del signo de igual.
`print`	`10 print w`	Mostrar el valor de `w`.
`goto`	`70 goto 45`	Transferir el control del programa a la línea `45`.
`if/goto`	`35 if i == z goto 80`	Comparar si `i` y `z` son iguales y transferir el control del programa a la línea `80` si la condición es verdadera; en caso contrario, continuar la ejecución con la siguiente instrucción.
`end`	`99 end`	Terminar la ejecución del programa.

Figura 17.22 | Comandos de Simple.

```
10 rem   determinar e imprimir la suma de dos enteros
15 rem
20 rem   introducir los dos enteros
30 input a
40 input b
45 rem
50 rem   sumar los enteros y almacenar el resultado en c
60 let c = a + b
65 rem
70 rem   imprimir el resultado
80 print c
90 rem   terminar la ejecución del programa
99 end
```

Figura 17.23 | Programa de Simple que determina la suma de dos enteros.

```
10 rem   determinar e imprimir el mayor de dos enteros
20 input s
30 input t
32 rem
35 rem   evaluar si s >= t
40 if s >= t goto 90
45 rem
50 rem   t es mayor que s, por lo que se imprime t
60 print t
70 goto 99
75 rem
80 rem   s es mayor o igual que t, por lo que se imprime s
90 print s
99 end
```

Figura 17.24 | Programa de Simple que encuentra el mayor de dos enteros.

Simple no cuenta con una instrucción de repetición (como las instrucciones `for`, `while` o `do...while` de Java). Sin embargo, Simple puede simular cada una de las instrucciones de repetición de Java mediante el uso de las instrucciones

`if/goto` y `goto`. En la figura 17.25 se utiliza un ciclo controlado por centinela para calcular los cuadrados de varios enteros. Cada entero se introduce desde el teclado y se almacena en la variable `j`. Si el valor introducido es el valor centinela `-9999`, el control se transfiere a la línea `99`, en donde termina el programa. En caso contrario, a `k` se le asigna el cuadrado de `j`, `k` se muestra en pantalla y el control se pasa a la línea `20`, en donde se introduce el siguiente entero.

Utilizando los programas de ejemplo de las figuras 17.23 a 17.25 como guía, escriba un programa de Simple para realizar cada una de las siguientes acciones:

a) Introducir tres enteros, determinar su promedio e imprimir el resultado.
b) Usar un ciclo controlado por centinela para introducir 10 enteros, calcular e imprimir su suma.
c) Usar un ciclo controlado por contador para introducir 7 enteros, algunos positivos y otros negativos, calcular e imprimir su promedio.
d) Introducir una serie de enteros, determinar e imprimir el mayor. El primer entero introducido indica cuántos números deben procesarse.
e) Introducir 10 enteros e imprimir el menor.
f) Calcular e imprimir la suma de los enteros pares del 2 al 30.
g) Calcular e imprimir el producto de los enteros impares del 1 al 9.

```
10 rem   calcular los cuadrados de varios enteros
20 input j
23 rem
25 rem   evaluar el valor centinela
30 if j == -9999 goto 99
33 rem
35 rem   calcular el cuadrado de j y asignar el resultado a k
40 let k = j * j
50 print k
53 rem
55 rem   iterar para obtener el siguiente valor de j
60 goto 20
99 end
```

Figura 17.25 | Calcular los cuadrados de varios enteros.

17.27 *(Construcción de un compilador; prerrequisitos: completar los ejercicios 7.34, 7.35, 17.12, 17.13 y 17.26)* Ahora que hemos presentado el lenguaje Simple (ejercicio 17.26), hablaremos sobre cómo construir un compilador de Simple. Primero debemos considerar el proceso mediante el cual un programa de Simple se convierte a LMS y se ejecuta por el simulador Simpletron (vea la figura 17.26). El compilador lee un archivo que contiene un programa de Simple y lo convierte en código de LMS. Este código se envía a un archivo en disco, en el que las instrucciones de LMS aparecen una en cada línea. Después el archivo de LMS se carga en el simulador Simpletron y los resultados se envían a un archivo en disco y a la pantalla. Observe que el programa de Simpletron desarrollado en el ejercicio 7.35 acepta su entrada mediante el teclado. Este programa debe modificarse para que lea desde un archivo y así pueda ejecutar los programas producidos por nuestro compilador.

El compilador de Simple realiza dos *pasadas* del programa de Simple para convertirlo en LMS. En la primera pasada se construye una *tabla de símbolos* (objeto) en la que cada *número de línea* (objeto), *nombre de variable* (objeto) y *constante* (objeto) del programa de Simple se guarda con su tipo y ubicación correspondiente en el código final de LMS (la tabla de símbolos se describe detalladamente a continuación). En la primera pasada también se produce(n) el (los) objeto(s) correspondientes a la instrucción de LMS para cada una de las instrucciones de Simple (objeto, etcétera). Si el programa de Simple contiene instrucciones que transfieren el control a una línea que se encuentra más adelante en el programa, la primera pasada produce un programa de LMS que contiene algunas instrucciones "no terminadas". En la segunda pasada del compilador se localizan y completan las instrucciones no terminadas, y se envía el programa de LMS a un archivo.

Primera pasada

El compilador empieza leyendo una instrucción del programa de Simple y la coloca en memoria. La línea debe separarse en sus *tokens* individuales (es decir, "piezas" de una instrucción) para su procesamiento y compilación. (Puede usarse la clase `StreamTokenizer` del paquete `java.io`). Recuerde que todas las instrucciones empiezan con un número de línea,

seguido de un comando. A medida que el compilador divide una instrucción en tokens, si éste es un número de línea, una variable o una constante, se coloca en la tabla de símbolos. Un número de línea se coloca en la tabla de símbolos solamente si es el primer token en una instrucción. El objeto `tablaDeSimbolos` es un arreglo de objetos `entradaTabla` que representan a cada uno de los símbolos en el programa. No hay restricción en cuanto al número de símbolos que pueden aparecer en el programa. Por lo tanto, la `tablaDeSimbolos` para un programa específico podría ser extensa. Por ahora, haga que la `tablaDeSimbolos` sea un arreglo de 100 elementos. Usted podrá incrementar o decrementar su tamaño una vez que el programa esté ejecutándose.

Cada objeto `entradaTabla` contiene tres campos. El campo `simbolo` es un entero que contiene la representación Unicode de una variable (recuerde que los nombres de las variables son caracteres individuales), un número de línea o una constante. El campo `tipo` es uno de los siguientes caracteres que indican el tipo de ese símbolo: `'C'` para constante, `'L'` para número de línea o `'V'` para variable. El campo `ubicacion` contiene la ubicación de memoria Simpletron (`00` a `99`) a la que hace referencia el símbolo. La memoria Simpletron es un arreglo de 100 enteros en donde se almacenan instrucciones y datos de LMS. Para un número de línea, la ubicación es el elemento en el arreglo de memoria Simpletron en el que empiezan las instrucciones de LMS para la instrucción de Simple. Para una variable o constante, la ubicación es el elemento en el arreglo de memoria Simpletron en el que se almacena esa variable o constante. Las variables y constantes se asignan desde el final de la memoria Simpletron hacia atrás. La primera variable o constante se almacena en la ubicación `99`, la siguiente en la ubicación `98` y así, sucesivamente.

La tabla de símbolos juega una parte integral para convertir los programas de Simple a LMS. En el capítulo 7 aprendimos que una instrucción de LMS es un entero de cuatro dígitos, compuesto de dos partes: el *código de la operación* y el *operando*. El código de operación se determina mediante los comandos en Simple. Por ejemplo, el comando `input` de Simple corresponde al código de operación `10` de LMS (lectura), y el comando `print` de Simple corresponde al código de operación `11` de LMS (escritura). El operando es una ubicación en memoria que contiene los datos sobre los cuales el código de operación lleva a cabo su tarea (por ejemplo, el código de operación `10` lee un valor desde el teclado y lo guarda en la ubicación de memoria especificada por el operando). El compilador busca en la `tablaDeSimbolos` para determinar la ubicación de memoria Simpletron para cada símbolo, de manera que pueda utilizarse la ubicación correspondiente para completar las instrucciones de LMS.

La compilación de cada instrucción de Simple se basa en su comando. Por ejemplo, después de insertar el número de línea de una instrucción `rem` en la tabla de símbolos, el compilador ignora el resto de la instrucción ya que un comentario es sólo para fines de documentación. Las instrucciones `input`, `print`, `goto` y `end` corresponden a las instrucciones *leer*, *escribir*, *bifurcar* (hacia una ubicación específica) y *parar* de LMS. Las instrucciones que contienen estos comandos de Simple se convierten directamente a LMS. (*Nota:* una instrucción `goto` puede contener una referencia no resuelta, si el número de línea especificado hace referencia a una instrucción que se encuentre más adelante en el archivo del programa de Simple; a esto se le conoce algunas veces como referencia adelantada).

Cuando una instrucción `goto` se compila con una referencia no resuelta, la instrucción de LMS debe *marcarse* para indicar que la segunda pasada del compilador debe completar la instrucción. Las banderas se almacenan en un arreglo de 100 elementos llamado `banderas` de tipo `int`, en donde cada elemento se inicializa con `-1`. Si la ubicación de memoria a la que hace referencia un número de línea en el programa de Simple no se conoce todavía (es decir, que no se encuentra en la tabla de símbolos), el número de línea se almacena en el arreglo `banderas`, en el elemento con el mismo índice que la instrucción incompleta. El operando de la instrucción incompleta se establece en `00` temporalmente. Por ejemplo, una instrucción de ramificación incondicional (que hace una referencia adelantada) se deja como `+4000` hasta la segunda pasada del compilador. En breve describiremos la segunda pasada del compilador.

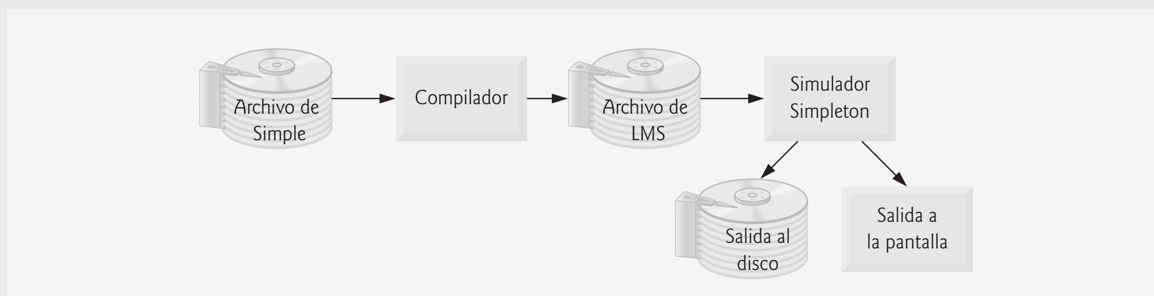


Figura 17.26 | Cómo escribir, compilar y ejecutar un programa en lenguaje Simple.

La compilación de las instrucciones `if/goto` y `let` es más complicada que las otras instrucciones; son las únicas instrucciones que producen más de una instrucción de LMS. Para una instrucción `if/goto`, el compilador produce código para evaluar la condición y ramificar hacia otra línea, en caso de ser necesario. El resultado de la ramificación podría ser una referencia no resuelta. Cada uno de los operadores relacionales y de igualdad pueden simularse mediante el uso de las instrucciones *branch zero* y *branch negative* de LMS (o posiblemente una combinación de ambas).

Para una instrucción `let`, el compilador produce código para evaluar una expresión aritmética arbitrariamente compleja que consiste de variables y/o constantes enteras. Las expresiones deben separar cada operando y operador con espacios. En los ejercicios 17.12 y 17.13 se presentaron el algoritmo de conversión de infijo a postfijo y el algoritmo de evaluación de expresiones postfijo que utilizan los compiladores para evaluar expresiones. Antes de proseguir con su compilador, debe completar cada uno de estos ejercicios. Cuando un compilador encuentra una expresión, la convierte de notación infijo a notación postfijo y después evalúa la expresión postfijo.

¿Cómo es que el compilador produce el lenguaje máquina para evaluar una expresión que contiene variables? El algoritmo de evaluación de expresiones postfijo contiene un "gancho" en donde el compilador puede generar instrucciones de LMS, en vez de evaluar la expresión. Para permitir este "gancho" en el compilador, el algoritmo de evaluación postfijo debe modificarse para buscar en la tabla de símbolos cada símbolo que vaya encontrando (y posiblemente insertarlo), determinar la ubicación de memoria correspondiente de ese símbolo y *meter la ubicación de memoria a la pila (en vez del símbolo)*. Cuando se encuentra un operador en la expresión postfijo, las dos ubicaciones de memoria que se encuentran en la parte superior de la pila se sacan y se produce el lenguaje máquina para llevar a cabo la operación, utilizando las ubicaciones de memoria como operandos. El resultado de cada subexpresión se almacena en una ubicación temporal en memoria y se mete de vuelta a la pila, de manera que pueda continuar la evaluación de la expresión postfijo. Al completarse esta evaluación, la ubicación de memoria que contiene el resultado es la única ubicación que queda en la pila. Esta ubicación se saca y se generan las instrucciones de LMS para asignar el resultado a la variable que está a la izquierda de la instrucción `let`.

Segunda pasada

En la segunda pasada del compilador se llevan a cabo dos tareas: Resolver cualquier referencia no resuelta y enviar el código de LMS a un archivo. La resolución de las referencias ocurre así:

a) Buscar en el arreglo `banderas` una referencia no resuelta (es decir, un elemento con un valor distinto de `-1`).
b) Localizar el objeto en el arreglo `tablaDeSimbolos` que contenga el símbolo almacenado en el arreglo `banderas` (asegúrese que el tipo del símbolo sea `'L'` para un número de línea).
c) Insertar la ubicación de memoria del campo `ubicacion` en la instrucción con la referencia no resuelta (recuerde que una instrucción que contiene una referencia no resuelta tiene el operando `00`).
d) Repetir los pasos *(a)*, *(b)* y *(c)* hasta que se llegue al final del arreglo `banderas`.

Una vez completo el proceso de resolución, todo el arreglo que contiene el código de LMS se envía a un archivo en disco, con una instrucción de LMS por línea. El simulador Simpletron puede leer este archivo para ejecutarlo (una vez que se modifique el simulador para que lea su entrada desde un archivo). El compilar su primer programa de Simple en un archivo de LMS y luego ejecutar ese archivo le dará un verdadero sentido de satisfacción personal.

Un ejemplo completo

En el siguiente ejemplo mostramos la conversión completa de un programa de Simple a LMS, como lo deberá realizar el compilador de Simple. Considere un programa de Simple que recibe un entero y suma los valores desde 1 hasta ese entero. El programa y las instrucciones de LMS producidas por la primera pasada del compilador de Simple se muestran en la figura 17.27 La tabla de símbolos construida por la primera pasada se muestra en la figura 17.28.

La mayoría de las instrucciones de Simple se convierten directamente a instrucciones de LMS individuales. Las excepciones en este programa son los comentarios, la instrucción `if/goto` en la línea `20` y las instrucciones `let`. Los comentarios no se traducen a lenguaje máquina. Sin embargo, el número de línea para un comentario se coloca en la tabla de símbolos, en caso de que se haga referencia al número de línea en una instrucción `goto` o `if/goto`. En la línea `20` del programa se especifica que, si la condición `y == x` es verdadera, el control del programa se transfiere a la línea `60`. Ya que la línea `60` aparece más adelante en el programa, la primera pasada del compilador todavía no ha colocado el valor `60` en la tabla de símbolos. (Se colocan números de línea en la tabla de símbolos solamente cuando aparecen como el primer token en una instrucción). Por lo tanto, no es posible en este momento determinar el operando de la instrucción *branch zero* de LMS que se encuentra en la ubicación `03` del arreglo de instrucciones de LMS. El compilador coloca `60` en la ubicación `03` del arreglo `banderas` para indicar que la instrucción va a completarse en la segunda pasada.

Programa de Simple	Ubicación e instrucción de LMS	Descripción
`5 rem  sumar 1 a x`	*ninguna*	`rem` se ignora
`10 input x`	00 +1099	leer x y colocar su valor en la ubicación 99
`15 rem verificar si y == x`	*ninguna*	`rem` se ignora
`20 if y == x goto 60`	01 +2098	cargar y (98) en el acumulador
	02 +3199	restar x (99) al acumulador
	03 +4200	ramificar a ubicación no resuelta si el resultado es cero
`25 rem   incrementar y`	*ninguna*	`rem` se ignora
`30 let y = y + 1`	04 +2098	cargar y en el acumulador
	05 +3097	sumar 1 (97) al acumulador
	06 +2196	almacenar en ubicación temporal 96
	07 +2096	cargar de ubicación temporal 96
	08 +2198	almacenar acumulador en y
`35 rem   sumar y al total`	*ninguna*	rem se ignora
`40 let t = t + y`	09 +2095	cargar t (95) en el acumulador
	10 +3098	sumar y al acumulador
	11 +2194	almacenar en ubicación temporal 94
	12 +2094	cargar de ubicación temporal 94
	13 +2195	almacenar acumulador en t
`45 rem   ciclo con y`	*ninguna*	`rem` se ignora
`50 goto 20`	14 +4001	ramificar a ubicación 01
`55 rem   mostrar resultado`	*ninguna*	`rem` se ignora
`60 print t`	15 +1195	mostrar t en la pantalla
`99 end`	16 +4300	terminar la ejecución

Figura 17.27 | Las instrucciones de LMS producidas después de la primera pasada del compilador.

Símbolo	Tipo	Ubicación
5	L	00
10	L	00
'x'	V	99
15	L	01
20	L	01
'y'	V	98
25	L	04
30	L	04
1	C	97
35	L	09

Figura 17.28 | Tabla de símbolos para el programa de la figura 17.27. (Parte 1 de 2).

Símbolo	Tipo	Ubicación
40	L	09
't'	V	95
45	L	14
50	L	14
55	L	15
60	L	15
99	L	16

Figura 17.28 | Tabla de símbolos para el programa de la figura 17.27. (Parte 2 de 2).

Debemos llevar el registro de la ubicación de la siguiente instrucción en el arreglo de LMS, ya que no hay una correspondencia de uno a uno entre las instrucciones de Simple y las instrucciones de LMS. Por ejemplo, la instrucción `if/goto` de la línea `20` se compila en tres instrucciones de LMS. Cada vez que se produce una instrucción, debemos incrementar el *contador de instrucciones* a la siguiente ubicación en el arreglo de LMS. Hay que tener en cuenta que el tamaño de la memoria Simpletron podría presentar un problema para los programas de Simple con muchas instrucciones, variables y constantes. Es posible que el compilador se quede sin memoria. Para probar este caso, su programa debe contener un *contador de datos* para llevar un registro de la ubicación en la que se almacenará la siguiente variable o constante en el arreglo de LMS. Si el valor del contador de instrucciones es mayor que el valor del contador de datos, significa que el arreglo de LMS está lleno. En este caso, el proceso de compilación debe terminar y el compilador debe imprimir un mensaje de error, indicando que se agotó la memoria durante la compilación. Esto sirve para enfatizar que, aunque el programador está libre de la carga que representa para el compilador tener que administrar la memoria, debe determinar cuidadosamente la colocación de instrucciones y datos en ella, y debe comprobar que no haya errores como el agotamiento de la memoria durante el proceso de compilación.

El proceso de compilación, paso a paso

Ahora analicemos el proceso de compilación para el programa de Simple que aparece en la figura 17.27. El compilador lee la primera línea del programa:

```
5 rem sumar 1 a x
```

en memoria. El primer token en la instrucción (el número de línea) se determina mediante el uso de la clase `StringTokenizer`. (En el capítulo 30 se describe el uso de esta clase). El token devuelto por el objeto `StringTokenizer` se convierte en un entero mediante el uso del método `static Integer.parseInt()`, de manera que el símbolo `5` puede localizarse en la tabla de símbolos. Si el símbolo no se encuentra, se inserta en la tabla de símbolos.

Como nos encontramos en el inicio del programa y ésta es la primera línea, todavía no hay símbolos en la tabla. Por lo tanto, se inserta el `5` en la tabla de símbolos con el tipo `L` (número de línea) y se le asigna la primera ubicación en el arreglo de LMS (`00`). Aunque esta línea es un comentario, de todas formas se asigna un espacio en la tabla de símbolos para el número de línea (en caso que se haga referencia al mismo en una instrucción `goto` o `if/goto`). No se genera ninguna instrucción de LMS para una instrucción `rem`, por lo que no se incrementa el contador de instrucciones. Ahora la instrucción:

```
10 input x
```

se divide en tokens. El número de línea `10` se coloca en la tabla de símbolo con el tipo `L` y se le asigna la primera ubicación en el arreglo de LMS (`00` pues, como un comentario empezó el programa, el contador de instrucciones sigue siendo `00`). El comando `input` indica que el siguiente token es una variable (sólo puede aparecer una variable en una instrucción `input`). Como `input` corresponde directamente a un código de operación de LMS, el compilador sólo tiene que determinar la ubicación de `x` en el arreglo de LMS. El símbolo `x` no se encuentra en la tabla de símbolos, por lo que se inserta en ésta como la representación Unicode de `x`, se le asigna el tipo `V` y la ubicación `99` en el arreglo de LMS (el almacenamiento de datos empieza en la ubicación `99` y se van asignando ubicaciones en forma regresiva). Ahora puede generarse el código LMS para esta instrucción. El código de operación `10` (código de operación de lectura de LMS)

se multiplica por 100 y se agrega la ubicación de x (según lo determinado en la tabla de símbolos) para completar la instrucción. Después la instrucción se almacena en el arreglo de LMS, en la ubicación 00. El contador de instrucciones se incrementa en uno, ya que se produjo una sola instrucción de LMS.

Ahora la instrucción:

```
15 rem   verificar si y == x
```

se divide en tokens. Se busca en la tabla de símbolos el número de línea 15 (el cual no se encuentra). El número de línea se inserta con el tipo L y se le asigna la siguiente ubicación en el arreglo, 01. (Recuerde que las instrucciones rem no producen código, por lo que no se incrementa el contador de instrucciones).

Ahora la instrucción:

```
20 if y == x goto 60
```

se divide en tokens. El número de línea 20 se inserta en la tabla de símbolos y se le asigna el tipo L en la siguiente ubicación en el arreglo de LMS (01). El comando if indica que se va a evaluar una condición. La variable y no se encuentra en la tabla de símbolos, por lo que se inserta, se le asigna el tipo V y la ubicación 98. A continuación se generan las instrucciones de LMS para evaluar la condición. Como no hay un equivalente directo en LMS para la instrucción if/goto; ésta debe simularse mediante un cálculo en el que se utilicen x y y, y después debe hacerse una bifurcación con base en el resultado. Si y es igual a x el resultado de restar x a y es cero, por lo que puede utilizarse la instrucción *branch zero* con el resultado del cálculo para simular la instrucción if/goto. El primer paso requiere que se cargue y (de la ubicación 98 del arreglo de LMS) en el acumulador. Esto produce la instrucción 01 +2098. Luego, se resta x del acumulador. Esto produce la instrucción 02 +3199. El valor en el acumulador puede ser cero, positivo o negativo. Como el operador es ==, queremos utilizar la operación *branch zero*. Primero se busca en la tabla de símbolos la ubicación de la ramificación (60 en este caso), la cual no se encuentra. Por lo tanto, 60 se coloca en el arreglo banderas en la ubicación 03, y se genera la instrucción 03 +4200. (No podemos agregar la ubicación de la ramificación, ya que todavía no hemos asignado una ubicación a la línea 60 en el arreglo de LMS). El contador de instrucciones se incrementa en 04.

El compilador continúa con la instrucción:

```
25 rem   incrementar y
```

El número de línea 25 se inserta en la tabla de símbolos con el tipo L y se le asigna la ubicación 04 en el arreglo de LMS. No se incrementa el contador de instrucciones.

Cuando la instrucción:

```
30 let y = y + 1
```

se divide en tokens, el número de línea 30 se inserta en la tabla de símbolos con el tipo L y se le asigna la ubicación 04 en el arreglo de LMS. El comando let indica que la línea es una instrucción de asignación. Primero se insertan todos los símbolos de la línea en la tabla de símbolos (si no es que están ya ahí). El entero 1 se agrega a la tabla de símbolos con el tipo C y se le asigna la ubicación 97 LMS. A continuación, el lado derecho de la asignación se convierte de notación infijo a postfijo. Luego se evalúa la expresión postfijo (y 1 +). El símbolo y se encuentra ya en la tabla de símbolos y su ubicación correspondiente en memoria se mete a la pila. El símbolo 1 también se encuentra ya en la tabla de símbolos y su ubicación correspondiente en memoria se mete a la pila. Al llegar al operador +, el evaluador de expresiones postfijo saca el elemento superior de la pila y lo coloca en el operando derecho del operador, saca de nuevo el elemento superior de la pila, lo coloca en el operando izquierdo del operador y produce las siguientes instrucciones de LMS:

```
04 +2098   (load y)
05 +3097   (add 1)
```

El resultado de la expresión se almacena en una ubicación temporal en memoria (96) mediante la instrucción:

```
06 +2196   (almacenar temporalmente)
```

y la ubicación temporal se mete en la pila. Ahora que se ha evaluado la expresión, el resultado debe almacenarse en y (es decir, la variable del lado izquierdo de =). Entonces la ubicación temporal se carga en el acumulador y éste se almacena en y mediante las instrucciones:

```
07 +2096   (cargar temporalmente)
08 +2198   (store y)
```

El lector observará inmediatamente que las instrucciones de LMS parecen ser redundantes. Hablaremos sobre esta cuestión en breve.

Cuando la instrucción:

```
35 rem   sumar y al total
```

se divide en tokens, el número de línea 35 se inserta en la tabla de símbolos con el tipo L y se le asigna la ubicación 09.

La instrucción:

```
40 let t = t + y
```

es similar a la línea 30. La variable t se inserta en la tabla de símbolos con el tipo V y se le asigna la ubicación 95 en el arreglo de LMS. Las instrucciones siguen la misma lógica y formato que la línea 30, y se generan las instrucciones 09 +2095, 10 +3098, 11 +2194, 12 +2094 y 13 +2195. Observe que el resultado de t + y se asigna a la ubicación temporal 94 antes de asignarse a t (95). Una vez más, el lector observará que las instrucciones en las ubicaciones de memoria 11 y 12 parecen ser redundantes. De nuevo, hablaremos sobre esta cuestión en breve.

La instrucción:

```
45 rem   ciclo con y
```

es un comentario, por lo que la línea 45 se agrega a la tabla de símbolos con el tipo L y se le asigna la ubicación 14 en el arreglo de LMS.

La instrucción:

```
50 goto 20
```

transfiere el control a la línea 20. El número de línea 50 se inserta en la tabla de símbolos con el tipo L y se le asigna la ubicación 14 en el arreglo de LMS. El equivalente de goto en LMS es la instrucción de *bifurcación incondicional* (40), la cual transfiere el control a una ubicación específica en el arreglo de LMS. El compilador busca en la tabla de símbolos la línea 20 y encuentra que a ésta le corresponde la ubicación 01 en el arreglo de LMS. El código de operación (40) se multiplica por 100 y se le agrega la ubicación 01 para producir la instrucción 14 +4001.

La instrucción:

```
55 rem   mostrar resultado
```

es un comentario, por lo que la línea 55 se inserta en la tabla de símbolos con el tipo L y se le asigna la ubicación 15 en el arreglo de LMS.

La instrucción:

```
60 print t
```

es una instrucción de salida. El número de línea 60 se inserta en la tabla de símbolos con el tipo L y se le asigna la ubicación 15 en el arreglo de LMS. El equivalente de print en LMS es el código de operación 11 (*write*). La ubicación de t se determina a partir de la tabla de símbolos y se agrega al resultado de la multiplicación del código de operación por 100.

La instrucción:

```
99 end
```

es la línea final del programa. El número de línea 99 se almacena en la tabla de símbolos con el tipo L y se le asigna la ubicación 16 en el arreglo de LMS. El comando end produce la instrucción de LMS +4300 (43 significa *halt* en LMS), la cual se escribe como instrucción final en el arreglo de memoria de LMS.

Esto completa la primera pasada del compilador. Ahora consideremos la segunda pasada. En el arreglo banderas se buscan valores distintos de -1. La ubicación 03 contiene 60, por lo que el compilador sabe que la instrucción 03 está incompleta. El compilador completa la instrucción buscando en la tabla de símbolos el número 60, determinando su ubicación y agregándola a la instrucción incompleta. En este caso la búsqueda determina que la línea 60 corresponde a la ubicación 15 en el arreglo de LMS, por lo que se produce la instrucción completa 03 +4215 que sustituye a 03 +4200. Ahora el programa de Simple se ha compilado con éxito.

Para crear el compilador, tendrá que llevar a cabo cada una de las siguientes tareas:

a) Modifique el programa simulador de Simpletron que escribió en el ejercicio 7.35 para que reciba la entrada de un archivo especificado por el usuario (vea el capítulo 14). El simulador debe mostrar sus resultados en un archivo en disco, con el mismo formato que el de la pantalla. Convierta el simulador para que sea un programa orientado a objetos. En especial, haga que cada parte del hardware sea un objeto. Ordene los tipos de instrucciones en una jerarquía de clases por medio de la herencia. Después ejecute el programa en forma polimórfica, indicando a cada instrucción que se ejecute a sí misma con un mensaje `ejecutarInstruccion`.

b) Modifique el algoritmo de conversión de expresiones infijo a postfijo del ejercicio 17.12 para procesar operandos enteros con varios dígitos y operandos de nombres de variables con una sola letra. [*Sugerencia:* puede utilizarse la clase `StringTokenizer` para localizar cada constante y variable en una expresión, y las constantes pueden convertirse de cadenas a enteros mediante el uso del método `parseInt` de la clase `Integer`]. [*Nota:* la representación de datos de la expresión postfijo debe alterarse para dar soporte a los nombres de variables y constantes enteras].

c) Modifique el algoritmo de evaluación de expresiones postfijo para procesar operandos enteros con varios dígitos y operandos de nombres de variables. Además, el algoritmo deberá ahora implementar el "gancho" que se describió anteriormente, de manera que se produzcan instrucciones de LMS en vez de evaluar directamente la expresión. [*Sugerencia:* puede utilizarse la clase `StringTokenizer` para localizar cada constante y variable en una expresión, y las constantes pueden convertirse de cadenas a enteros mediante el uso del método `parseInt` de la clase `Integer`]. [*Nota:* la representación de datos de la expresión postfijo debe alterarse para dar soporte a los nombres de variables y constantes enteras].

d) Construya el compilador. Incorpore las partes b) y c) para evaluar las expresiones en instrucciones `let`. Su programa debe contener un método que realice la primera pasada del compilador y un método que realice la segunda pasada del compilador. Ambos métodos pueden llamar a otros métodos para realizar sus tareas. Haga que su compilador esté lo más orientado a objetos que sea posible.

17.28 *(Optimización del compilador de Simple)* Cuando se compila un programa y se convierte en LMS, se genera un conjunto de instrucciones. Ciertas combinaciones de instrucciones a menudo se repiten, por lo general, en tercias conocidas como *producciones*. Una producción normalmente consiste de tres instrucciones tales como *load*, *add* y *store*. Por ejemplo, en la figura 17.29 se muestran cinco de las instrucciones de LMS que se produjeron en la compilación del programa de la figura 17.27. Las primeras tres instrucciones son la producción que suma `1` a `y`. Observe que las instrucciones `06` y `07` almacenan el valor del acumulador en la ubicación temporal `96` y cargan el valor de vuelta en el acumulador, de manera que la instrucción `08` pueda almacenar el valor en la ubicación `98`. A menudo una producción va seguida de una instrucción *load* para la misma ubicación en la que fue guardada. Este código puede *optimizarse* mediante la eliminación de la instrucción *store* y la instrucción *load* que le sigue, las cuales operan en la misma ubicación, con lo que se permite al simulador Simpletron ejecutar el programa con más rapidez. En la figura 17.30 se muestra el LMS optimizado para el programa de la figura 17.27. Observe que hay cuatro instrucciones menos en el código optimizado; un ahorro de espacio en memoria del 25%.

```
04    +2098    (load)
05    +3097    (add)
06    +2196    (store)
07    +2096    (load)
08    +2198    (store)
```

Figura 17.29 | Código sin optimizar del programa de la figura 17.27.

Programa de Simple	Ubicación e instrucción de LMS	Descripción
`5 rem  sumar 1 a x`	*ninguna*	`rem` se ignora
`10 input x`	`00 +1099`	leer `x` y colocar su valor en la ubicación `99`
`15 rem   verificar si y == x`	*ninguna*	`rem` se ignora

Figura 17.30 | Código optimizado para el programa de la figura 17.27. (Parte 1 de 2).

Programa de Simple	Ubicación e instrucción de LMS	Descripción
`20 if y == x goto 60`	`01 +2098`	cargar `y (98)` en el acumulador
	`02 +3199`	restar `x (99)` al acumulador
	`03 +4211`	ramificar a ubicación `11` si vale cero
`25 rem   incrementar y`	*ninguna*	`rem` se ignora
`30 let y = y + 1`	`04 +2098`	cargar `y` en el acumulador
	`05 +3097`	sumar `1 (97)` al acumulador
	`06 +2198`	almacenar acumulador en `y (98)`
`35 rem   sumar y al total`	*ninguna*	rem se ignora
`40 let t = t + y`	`07 +2096`	cargar `t` de la ubicación `(96)`
	`08 +3098`	sumar `y (98)` al acumulador
	`09 +2196`	almacenar acumulador en `t (96)`
`45 rem   ciclo con y`	*ninguna*	`rem` se ignora
`50 goto 20`	`10 +4001`	ramificar a ubicación `01`
`55 rem   mostrar resultado`	*ninguna*	`rem` se ignora
`60 print t`	`11 +1195`	mostrar `t (96)` en la pantalla
`99 end`	`12 +4300`	terminar la ejecución

Figura 17.30 | Código optimizado para el programa de la figura 17.27. (Parte 2 de 2).

17.29 *(Modificaciones al compilador de Simple)* Realice las siguientes modificaciones al compilador de Simple. Algunas de estas modificaciones podrían requerir también que se modifique el programa simulador de Simpletron que se escribió en el ejercicio 7.35.

a) Permitir el uso del operador residuo (`%`) en instrucciones `let`. El Lenguaje Máquina Simpletron debe modificarse para incluir una instrucción residuo.
b) Permitir la exponenciación en una instrucción `let` mediante el uso de `^` como operador de exponenciación. El Lenguaje Máquina Simpletron debe modificarse para incluir una instrucción de exponenciación.
c) Permitir al compilador que reconozca letras mayúsculas y minúsculas en instrucciones de Simple (por ejemplo, `'A'` es equivalente a `'a'`). No se requieren modificaciones al simulador de Simpletron.
d) Permitir que las instrucciones `input` lean valores para múltiples variables, como `input x, y`. No se requieren modificaciones al simulador Simpletron para llevar a cabo esta mejora en el compilador de Simple.
e) Permitir que el compilador muestre múltiples valores en una sola instrucción `print` como `print a, b, c`. No se requieren modificaciones al simulador de Simpletron para llevar a cabo esta mejora.
f) Agregar al compilador la capacidad de comprobar la sintaxis, de manera que se muestren mensajes de error cuando se encuentren errores de sintaxis en un programa de Simple. No se requieren modificaciones al simulador de Simpletron.
g) Permitir arreglos de enteros. No se requieren modificaciones al simulador de Simpletron para llevar a cabo esta mejora.
h) Permitir subrutinas especificadas por los comandos `gosub` y `return` de Simple. El comando `gosub` pasa el control del programa a una subrutina y el comando `return` pasa el control de vuelta a la instrucción que va después de `gosub`. Esto es similar a la llamada a un método en Java. La misma subrutina puede llamarse desde muchos comandos `gosub` distribuidos a lo largo de un programa. No se requieren modificaciones al simulador de Simpletron.
i) Permitir instrucciones de repetición de la forma:

```
for x = 2 to 10 step 2
    instrucciones de Simple
next
```

Esta instrucción `for` realiza iteraciones del `2` al `10` con un incremento de `2`. La línea `next` indica el final del cuerpo de la instrucción `for`. No se requieren modificaciones al simulador de Simpletron.

j) Permitir instrucciones de repetición de la forma:

```
for x = 2 to 10
     instrucciones de Simple
next
```

Esta instrucción `for` realiza iteraciones del `2` al `10` con un incremento predeterminado de `1`. No se requieren modificaciones al simulador de Simpletron.

k) Permitir que el compilador procese operaciones de entrada y salida con cadenas. Para ello se requiere la modificación del simulador Simpletron para que procese y almacene valores de cadena. [*Sugerencia:* cada palabra de Simpletron (es decir, ubicación de memoria) puede dividirse en dos grupos, cada uno de los cuales almacena un entero de dos dígitos. Cada entero de dos dígitos representa el equivalente decimal Unicode de un carácter. Agregue una instrucción de lenguaje máquina que imprima una cadena, empezando en cierta ubicación de memoria Simpletron. La primera mitad de la palabra Simpletron en esa ubicación es un conteo del número de caracteres en la cadena (es decir, la longitud de la misma). Cada mitad de palabra subsiguiente contiene un carácter Unicode expresado mediante dos dígitos decimales. La instrucción de lenguaje máquina comprueba la longitud e imprime la cadena, traduciendo cada número de dos dígitos en su carácter equivalente].

l) Permitir que el compilador procese valores de punto flotante, además de enteros. El simulador Simpletron también debe modificarse para procesar valores de punto flotante.

17.30 *(Un intérprete de Simple)* Un intérprete es un programa que lee la instrucción de un programa escrito en un lenguaje de alto nivel, determina la operación que va a realizar esa instrucción y la ejecuta inmediatamente. El programa en lenguaje de alto nivel no se convierte primero en lenguaje máquina. Los intérpretes se ejecutan con más lentitud que los compiladores, ya que cada instrucción que se encuentra en el programa que va a interpretarse debe primero descifrarse en tiempo de ejecución. Si las instrucciones están contenidas dentro de un ciclo, se descifran cada vez que se encuentran en éste. Las primeras versiones del lenguaje de programación BASIC se implementaron como intérpretes. La mayoría de los programas de Java se ejecutan mediante un intérprete.

Escriba un intérprete para el lenguaje Simple descrito en el ejercicio 17.26. El programa debe utilizar el convertidor de expresiones infijo a postfijo que se desarrolló en el ejercicio 17.12, junto con el evaluador de expresiones postfijo que se desarrolló en el ejercicio 17.13, para evaluar las expresiones en una instrucción `let`. Las mismas restricciones impuestas sobre el lenguaje Simple en el ejercicio 17.26 se aplican también a este programa. Pruebe el intérprete con los programas de Simple que se escribieron en el ejercicio 17.26. Compare los resultados de ejecutar estos programas en el intérprete con los resultados de compilar los programas de Simple y ejecutarlos en el simulador Simpletron que se construyó en el ejercicio 7.35.

17.31 *(Insertar/eliminar en cualquier parte de una lista enlazada)* Nuestra clase de lista enlazada permite inserciones y eliminaciones sólo en la parte frontal y en la parte posterior de la lista enlazada. Estas capacidades eran convenientes para nosotros cuando utilizamos la herencia o la composición para producir una clase pila y una clase cola con una mínima cantidad de código, con sólo reutilizar la clase lista. Normalmente, las listas enlazadas son más generales que las que nosotros vimos. Modifique la clase lista enlazada que desarrollamos en este capítulo para permitir inserciones y eliminaciones en cualquier parte de la lista.

17.32 *(Listas y colas sin referencias a la parte final)* En nuestra implementación de una lista enlazada (figura 17.3) utilizamos un `primerNodo` y un `ultimoNodo`. El `ultimoNodo` es útil para los métodos `insertarAlFinal` y `eliminarDelFinal` de la clase `Lista`. El método `insertarAlFinal` corresponde al método `enqueue` de la clase `Cola`.

Vuelva a escribir la clase `Lista` de manera que no utilice un `ultimoNodo`. De esta forma, cualquier operación en la parte final de la lista deberá empezar buscando en la lista desde su parte inicial. ¿Afecta esto a nuestra implementación de la clase `Cola` (figura 17.13)?

17.33 *(Rendimiento de los procesos de ordenamiento y búsqueda en árboles binarios)* Un problema con el ordenamiento de árboles binarios es que el orden en el que se insertan los datos afecta a la forma del árbol; para la misma colección de datos, distintos ordenamientos pueden producir árboles binarios de formas considerablemente distintas. El rendimiento de los algoritmos de ordenamiento y búsqueda en árboles binarios es susceptible a la forma del árbol binario. ¿Qué forma tendría un árbol binario si sus datos se insertaran en orden ascendente?, ¿en orden descendente?, ¿qué forma debería tener el árbol para lograr un máximo rendimiento en el proceso de búsqueda?

17.34 *(Listas indizadas)* Como se presentan en el texto, la búsqueda en las listas enlazadas debe llevarse a cabo en forma secuencial. Para las listas extensas, esto puede ocasionar un rendimiento pobre. Una técnica común para mejorar el rendimiento del proceso de búsqueda en las listas es crear y mantener un índice de la lista. Un índice es un conjunto de referencias a lugares clave en la lista. Por ejemplo, una aplicación que busca en una lista extensa de nombres podría mejorar el rendimiento al crear un índice con 26 entradas: una para cada letra del alfabeto. Una operación de búsqueda para un apellido que empiece con 'Y' buscaría entonces primero en el índice para determinar en dónde empiezan las entradas con 'Y' y luego "saltaría" hasta ese punto en la lista para buscar linealmente hasta encontrar el nombre deseado. Esto sería mucho más rápido que buscar en la lista enlazada desde el principio. Utilice la clase `Lista` de la figura 17.3 como la base para una clase `ListaIndizada`.

Escriba un programa que demuestre la operación de las listas indizadas. Asegúrese de incluir los métodos `insertarEnListaIndizada`, `buscarEnListaIndizada` y `eliminarDeListaIndizada`.

17.35 En la sección 17.7 creamos una clase de pila a partir de la clase `Lista` mediante la herencia (figura 17.10) y la composición (figura 17.12). En la sección 17.8 creamos una clase de cola a partir de la clase `Lista` mediante la composición (figura 17.13). Cree una clase de cola que herede de la clase `Lista`. ¿Cuáles son las diferencias entre esta clase y la que creamos con la composición?

18



Todo hombre de genio ve el mundo desde un ángulo distinto al de sus semejantes.

—Havelock Ellis

…nuestra individualidad especial, según se distingue desde nuestra genérica humanidad.

—Oliver Wendell Holmes, Sr.

Nacido bajo una ley, hacia otro límite.

—Lord Brooke

Trata con el material crudo de la opinión y, si mis convicciones tienen validez alguna, la opinión en última instancia gobierna al mundo.

—Woodrow Wilson

Genéricos

OBJETIVOS

En este capítulo aprenderá a:

- Crear métodos genéricos que realicen tareas idénticas en argumentos de distintos tipos.
- Crear una clase `Pila` genérica, que puede usarse para almacenar objetos de cualquier clase o tipo de interfaz.
- Comprender cómo sobrecargar los métodos genéricos con métodos no genéricos, o con otros métodos genéricos.
- Comprender los tipos crudos (raw) y cómo ayudan a lograr la compatibilidad inversa.
- Utilizar comodines cuando no se requiere información precisa sobre los tipos en el cuerpo de un método.
- Comprender la relación entre los genéricos y la herencia.

Plan general

18.1 Introducción

Sería bueno si pudiéramos escribir un solo método `ordenar` que pudiera ordenar los elementos en un arreglo `Integer`, en un arreglo `String` o en cualquier tipo que soporte el ordenamiento (es decir, que sus elementos puedan compararse). También sería bueno si pudiéramos escribir una sola clase `Pila` que pudiera utilizarse como `Pila` de enteros, de números de punto flotante, de objetos `String`, o una `Pila` de cualquier otro tipo. Sería aún mejor si pudiéramos detectar errores en los tipos en tiempo de compilación; a esto se le conoce como **seguridad de los tipos en tiempo de compilación**. Por ejemplo, si una `Pila` sólo almacena enteros, y trata de meter un objeto `String` en esa `Pila`, se produciría un error en tiempo de compilación.

Este capítulo habla sobre los **genéricos**, que proporcionan los medios para crear los modelos generales antes mencionados. Los **métodos genéricos** y las **clases genéricas** permiten a los programadores especificar, con la declaración de un solo método, un conjunto de métodos relacionados o, con la declaración de una sola clase, un conjunto de tipos relacionados, respectivamente. Los genéricos también proporcionan una seguridad de los tipos en tiempo de compilación, la cual permite a los programadores atrapar tipos inválidos en tiempo de compilación.

Podríamos escribir un método genérico para ordenar un arreglo de objetos, y después invocar el método genérico con arreglos `Integer`, arreglos `Double`, arreglos `String` y así en lo sucesivo, para ordenar los elementos del arreglo. El compilador podría realizar la comprobación de tipos para asegurar que el arreglo que se pasa al método para ordenar contenga elementos con el mismo tipo. Podríamos escribir una sola clase `Pila` genérica para manipular una pila de objetos, y después instanciar objetos `Pila` para una pila de objetos `Integer`, una pila de objetos `Double`, una pila de objetos `String`, y así en lo sucesivo. El compilador podría realizar la comprobación de tipos para asegurar que la `Pila` almacene elementos del mismo tipo.



Observación de ingeniería de software 18.1

Los métodos genéricos y las clases genéricas son de las características más poderosas de Java para la reutilización de software, con seguridad de los tipos en tiempo de compilación.

En este capítulo se presentan ejemplos de métodos genéricos y clases genéricas. También se consideran las relaciones entre los genéricos y otras características de Java, como la sobrecarga y la herencia. El capítulo 19, Colecciones, presenta un tratamiento detallado acerca de los métodos y clases genéricas del Marco de trabajo Collections de Java. Este marco de trabajo utiliza los genéricos para permitir a los programadores especificar los tipos exactos de objetos que una colección específica almacenará en un programa.

18.2 Motivación para los métodos genéricos

A menudo, los métodos sobrecargados se utilizan para realizar operaciones similares en distintos tipos de datos. Para motivar los métodos genéricos, empecemos con un ejemplo (figura 18.1) que contiene tres métodos `imprimirArreglo` sobrecargados (líneas 7 a 14, líneas 17 a 24 y líneas 27 a 34). Estos métodos imprimen las representaciones

de cadena de los elementos de un arreglo `Integer`, un arreglo `Double` y un arreglo `Character`, respectivamente. Observe que pudimos haber utilizado arreglos de los tipos primitivos `int`, `double` y `char` en este ejemplo. Optamos por usar arreglos de tipo `Integer`, `Double` y `Character` para establecer nuestro ejemplo de un método genérico, porque sólo se pueden usar tipos de referencias con los métodos y las clases genéricas.

Para empezar, el programa declara e inicializa tres arreglos: un arreglo `Integer` de seis elementos llamado `arregloInteger` (línea 39), un arreglo `Double` de siete elementos llamado `arregloDouble` (línea 40) y un arre-

```
1  // Fig. 18.1: MetodosSobrecargados.java
2  // Uso de métodos sobrecargados para imprimir arreglos de distintos tipos.
3
4  public class MetodosSobrecargados
5  {
6     // método imprimirArreglo para imprimir arreglo Integer
7     public static void imprimirArreglo( Integer[] arregloEntrada )
8     {
9        // muestra los elementos del arreglo
10       for ( Integer elemento : arregloEntrada )
11          System.out.printf( "%s ", elemento );
12
13       System.out.println();
14    } // fin del método imprimirArreglo
15
16    // método imprimirArreglo para imprimir arreglo Double
17    public static void imprimirArreglo( Double[] arregloEntrada )
18    {
19       // muestra los elementos del arreglo
20       for ( Double elemento : arregloEntrada )
21          System.out.printf( "%s ", elemento );
22
23       System.out.println();
24    } // fin del método imprimirArreglo
25
26    // método imprimirArreglo para imprimir arreglo Character
27    public static void imprimirArreglo( Character[] arregloEntrada )
28    {
29       // muestra los elementos del arreglo
30       for ( Character elemento : arregloEntrada )
31          System.out.printf( "%s ", elemento );
32
33       System.out.println();
34    } // fin del método imprimirArreglo
35
36    public static void main( String args[] )
37    {
38       // crea arreglos de objetos Integer, Double y Character
39       Integer[] arregloInteger = { 1, 2, 3, 4, 5, 6 };
40       Double[] arregloDouble = { 1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7 };
41       Character[] arregloCharacter = { 'H', 'O', 'L', 'A' };
42
43       System.out.println( "El arreglo arregloInteger contiene:" );
44       imprimirArreglo( arregloInteger ); // pasa un arreglo Integer
45       System.out.println( "\nEl arreglo arregloDouble contiene:" );
46       imprimirArreglo( arregloDouble ); // pasa un arreglo Double
47       System.out.println( "\nEl arreglo arregloCharacter contiene:" );
48       imprimirArreglo( arregloCharacter ); // pasa un arreglo Character
49    } // fin de main
50 } // fin de la clase MetodosSobrecargados
```

Figura 18.1 | Impresión de los elementos de un arreglo mediante el uso de métodos sobrecargados. (Parte 1 de 2).

```
El arreglo arregloInteger contiene:
1 2 3 4 5 6

El arreglo arregloDouble contiene:
1.1 2.2 3.3 4.4 5.5 6.6 7.7

El arreglo arregloCharacter contiene:
H O L A
```

Figura 18.1 | Impresión de los elementos de un arreglo mediante el uso de métodos sobrecargados. (Parte 2 de 2).

glo `Character` de cuatro elementos llamado `arregloCharacter` (línea 41). Después, en las líneas 43 a 48 se imprimen los arreglos.

Cuando el compilador encuentra la llamada a un método, siempre trata de localizar la declaración de un método que tenga el mismo nombre y parámetros que coincidan con los tipos de los argumentos en la llamada al método. En este ejemplo, cada llamada a `imprimirArreglo` coincide exactamente con una de las declaraciones del método `imprimirArreglo`. Por ejemplo, en la línea 44 se hace una llamada a `imprimirArreglo` con `arregloInteger` como argumento. En tiempo de compilación, el compilador determina el tipo del argumento `arregloInteger` (es decir, `Integer[]`) y trata de localizar un método llamado `imprimirArreglo` que especifique un solo parámetro `Integer[]` (líneas 7 a 14), y establece una llamada a ese método. De manera similar, cuando el compilador encuentra la llamada a `imprimirArreglo` en la línea 46, determina el tipo del argumento `arregloDouble` (es decir, `Double[]`), después trata de localizar un método llamado `imprimirArreglo` que especifique un solo parámetro `Double[]` (líneas 17 a 24) y establece una llamada a ese método. Por último, cuando el compilador encuentra la llamada a `imprimirArreglo` en la línea 48, determina el tipo del argumento `arregloCharacter` (es decir, `Carácter []`), después trata de localizar un método llamado `imprimirArreglo` que especifique un solo parámetro `Character[]` (líneas 27 a 34) y establece una llamada a ese método.

Estudie cada método `imprimirArreglo`. Observe que el tipo de los elementos del arreglo aparece en dos ubicaciones en cada método: el encabezado del método (líneas 7, 17 y 27) y el encabezado de la instrucción `for` (líneas 10, 20 y 30). Si reemplazáramos los tipos de los elementos en cada método con un nombre genérico (por convención, usaremos `E` para representar el tipo "elemento"), entonces los tres métodos se verían como el de la figura 18.2. Sucede que, si podemos reemplazar el tipo de los elementos del arreglo en cada uno de los tres métodos con un solo tipo genérico, entonces debemos poder declarar un método `imprimirArreglo` que pueda mostrar las representaciones de cadena de los elementos de un arreglo que contiene objetos. Observe que podemos utilizar el especificador de formato `%s` para imprimir la representación de cadena de cualquier objeto; se hará una llamada implícita al método `toString` del objeto. El método de la figura 18.2 es similar a la declaración del método `imprimirArreglo` genérico que vimos en la sección 18.3.

```
public static void imprimirArreglo( E[] arregloEntrada )
{
   // muestra los elementos del arreglo
   for ( E elemento : arregloEntrada )
      System.out.printf( "%s ", elemento );

   System.out.println();
} // fin del método imprimirArreglo
```

Figura 18.2 | Método `imprimirArreglo` en el que los nombres reales de los tipos se reemplazan por convención con el nombre genérico `E`.

18.3 Métodos genéricos: implementación y traducción en tiempo de compilación

Si las operaciones realizadas por varios métodos sobrecargados son idénticas para cada tipo de argumento, los métodos sobrecargados pueden codificarse en forma más compacta y conveniente, mediante el uso de un método

genérico. Puede escribir la declaración de un solo método genérico que pueda llamarse con argumentos de distintos tipos. Con base en los tipos de los argumentos que se pasan al método genérico, el compilador maneja cada llamada al método de manera apropiada.

En la figura 18.3 se vuelve a implementar la aplicación de la figura 18.1, usando un método `imprimirArreglo` genérico (líneas 7 a 14). Observe que las llamadas al método `imprimirArreglo` en las líneas 24, 26 y 28 son idénticas a las de la figura 18.1 (líneas 44, 46 y 48), y que los resultados de las dos aplicaciones son idénticos. Esto demuestra considerablemente el poder expresivo de los genéricos.

En la línea 7 empieza la declaración del método `imprimirArreglo`. Todas las declaraciones de métodos genéricos tienen una **sección de parámetros de tipo**, delimitada por **signos** < y > que se anteponen al tipo de valor de retorno del método (en este ejemplo, `< E >`). Cada sección de parámetro de tipo contiene uno o más **parámetros de tipo** (también llamados **parámetros de tipo formal**), separados por comas. Un parámetro de tipo, también conocido como **variable de tipo**, es un identificador que especifica el nombre de un tipo genérico. Los parámetros de tipo se pueden utilizar para declarar el tipo de valor de retorno, los tipos de los parámetros y los tipos de las variables locales en la declaración de un método genérico, y actúan como receptáculos para los tipos de los argumentos que se pasan al método genérico, que conocemos como **argumentos de tipos actuales**. El cuerpo de un método genérico se declara como el de cualquier otro método. Observe que los parámetros de tipo sólo

```
1  // Fig. 18.3: PruebaMetodoGenerico.java
2  // Uso de métodos genéricos para imprimir arreglos de distintos tipos.
3
4  public class PruebaMetodoGenerico
5  {
6     // método genérico imprimirArreglo
7     public static < E > void imprimirArreglo( E[] arregloEntrada )
8     {
9        // muestra los elementos del arreglo
10       for ( E elemento : arregloEntrada )
11          System.out.printf( "%s ", elemento );
12
13       System.out.println();
14    } // fin del método imprimirArreglo
15
16    public static void main( String args[] )
17    {
18       // crea arreglos de objetos Integer, Double y Character
19       Integer[] arregloInteger = { 1, 2, 3, 4, 5, 6 };
20       Double[] arregloDouble = { 1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7 };
21       Character[] arregloCharacter = { 'H', 'O', 'L', 'A' };
22
23       System.out.println( "El arreglo arregloInteger contiene:" );
24       imprimirArreglo( arregloInteger ); // pasa un arreglo Integer
25       System.out.println( "\nEl arreglo arregloDouble contiene:" );
26       imprimirArreglo( arregloDouble ); // pasa un arreglo Double
27       System.out.println( "\nEl arreglo arregloCharacter contiene:" );
28       imprimirArreglo( arregloCharacter ); // pasa un arreglo Character
29    } // fin de main
30 } // fin de la clase PruebaMetodoGenerico
```

```
El arreglo arregloInteger contiene:
1 2 3 4 5 6

El arreglo arregloDouble contiene:
1.1 2.2 3.3 4.4 5.5 6.6 7.7

El arreglo arregloCharacter contiene:
H O L A
```

Figura 18.3 | Impresión de los elementos de un arreglo, usando el método genérico `imprimirArreglo`.

pueden representar tipos por referencia, y no tipos primitivos (como `int`, `double` y `char`). Observe también que los nombres de los parámetros de tipo en la declaración del método deben coincidir con los que están declarados en la sección de parámetros de tipo. Por ejemplo, en la línea 10 se declara `elemento` como de tipo `E`, lo cual coincide con el parámetro de tipo (`E`) declarado en la línea 7. Además, un parámetro de tipo puede declararse sólo una vez en la sección de parámetros de tipo, pero puede aparecer más de una vez en la lista de parámetros del método. Por ejemplo, el nombre del parámetro de tipo `E` aparece dos veces en la siguiente lista de parámetros del método:

```
public static < E > void imprimirDosArreglos( E[] arreglo1, E[] arreglo2 )
```

Los parámetros de tipo no necesitan ser únicos entre los distintos métodos genéricos.



Error común de programación 18.1

Al declarar un método genérico, si no se coloca una sección de parámetros de tipo antes del tipo de valor de retorno de un método, se produce un error de sintaxis; el compilador no comprenderá el nombre del parámetro de tipo cuando lo encuentre en el método.

La sección de parámetros de tipo del método `imprimirArreglo` declara el parámetro de tipo `E` como el receptáculo para el tipo de los elementos del arreglo que imprimirá `imprimirArreglo`. Observe que `E` aparece en la lista de parámetros como el tipo de los elementos del arreglo (línea 7). El encabezado de la instrucción `for` (línea 10) también utiliza a `E` como el tipo de los elementos. Éstas son las mismas dos ubicaciones en las que los métodos sobrecargados `imprimirArreglo` de la figura 18.1 especificaron a `Integer`, `Double` o `Character` como el tipo de los elementos del arreglo. El resto de `imprimirArreglo` es idéntico a las versiones que se presentan en la figura 18.1.



Buena práctica de programación 18.1

Se recomienda especificar los parámetros de tipo como letras mayúsculas individuales. Por lo general, un parámetro que representa el tipo de los elementos de un arreglo (o cualquier otra colección) se llama `E`, en representación de "elemento".

Al igual que en la figura 18.1, el programa empieza por declarar e inicializar el arreglo `Integer` de seis elementos llamado `arregloInteger` (línea 19), el arreglo `Double` de siete elementos llamado `arregloDouble` (línea 20) y el arreglo `Character` de cuatro elementos llamado 7 (línea 21). Después el programa imprime cada arreglo mediante una llamada a `imprimirArreglo` (líneas 24, 26 y 28): una vez con el argumento `arregloInteger`, una vez con el argumento `arregloDouble` y una vez con el argumento `arregloCharacter`.

Cuando el compilador encuentra la línea 24, primero determina el tipo del argumento de `arregloInteger` (es decir, `Integer[]`) y trata de localizar un método llamado `imprimirArreglo`, el cual especifica un solo parámetro `Integer[]`. No hay un método así en este ejemplo. Después, el compilador determina si hay un método genérico llamado `imprimirArreglo`, el cual especifica un solo parámetro para el arreglo y utiliza un parámetro de tipo para representar el tipo de los elementos del arreglo. El compilador determina que `imprimirArreglo` (líneas 7 a 14) es una coincidencia y establece una llamada a ese método. El mismo proceso se repite para las llamadas al método `imprimirArreglo` en las líneas 26 y 28.



Error común de programación 18.2

Si el compilador no puede relacionar una llamada a un método con una declaración de método no genérico o genérico, se produce un error de compilación.



Error común de programación 18.3

Si el compilador no encuentra la declaración de un método que coincida exactamente con una llamada a un método, pero encuentra dos o más métodos genéricos que puedan satisfacer la llamada a ese método, se produce un error de compilación.

Además de establecer las llamadas a los métodos, el compilador también determina si las operaciones en el cuerpo del método se pueden aplicar a los elementos del tipo almacenado en el argumento del arreglo. La única operación que se realiza con los elementos del arreglo en este ejemplo es imprimir la representación de cadena de los elementos. En la línea 11 se realiza una llamada implícita a `toString` en cada `elemento`. Para trabajar

con los genéricos, cada elemento del arreglo debe ser un objeto de una clase o tipo de interfaz. Como todos los objetos tienen un método `toString`, el compilador está satisfecho de que en la línea 11 se realice una operación válida para cualquier objeto en el argumento arreglo de `imprimirArreglo`. Los métodos `toString` de las clases `Integer`, `Double` y `Character` devuelven la representación de cadena del valor `int`, `double` o `char` subyacente, respectivamente.

Cuando el compilador traduce el método genérico `imprimirArreglo` en códigos byte de Java, elimina la sección de parámetros de tipo y reemplaza los parámetros de tipo con tipos reales. A este proceso se le conoce como **borrado**. De manera predeterminada, todos los tipos genéricos se reemplazan con el tipo `Object`. Por lo tanto, la versión compilada del método `imprimirArreglo` aparece como se muestra en la figura 18.4; sólo hay una copia de este código que se utiliza para todas las llamadas a `imprimirArreglo` en el ejemplo. Esto es bastante distinto de otros mecanismo similares, como las plantillas de C++, en las cuales se genera y se compila una copia separada del código fuente para cada tipo que se pasa como argumento al método. Como veremos en la sección 18.4, la traducción y compilación de los genéricos es un proceso un poco más complicado de lo que hemos visto en esta sección.

Al declarar a `imprimirArreglo` como método genérico en la figura 18.3, eliminamos la necesidad de los métodos sobrecargados de la figura 18.1, ahorrando 20 líneas de código y creando un método reutilizable que pueda imprimir las representaciones de cadena de los elementos en cualquier arreglo que contenga objetos. Sin embargo, este ejemplo en especial pudo haber declarado simplemente el método `imprimirArreglo` como se muestra en la figura 18.4, usando un arreglo `Object` como parámetro. Esto habría producido los mismos resultados, ya que cualquier objeto `Object` se puede imprimir como objeto `String`. En un método genérico, los beneficios se hacen presentes cuando el método también utiliza un parámetro de tipo como el tipo de valor de retorno del método, como lo demostraremos en la siguiente sección.

```
public static void imprimirArreglo( Object[] arregloEntrada )
{
   // muestra los elementos del arreglo
   for ( Object elemento : arregloEntrada )
      System.out.printf( "%s ", elemento );

   System.out.println();
} // fin del método imprimirArreglo
```

Figura 18.4 | El método genérico `imprimirArreglo`, una vez que el compilador realiza el proceso de borrado.

18.4 Cuestiones adicionales sobre la traducción en tiempo de compilación: métodos que utilizan un parámetro de tipo como tipo de valor de retorno

Consideremos un ejemplo de método genérico, en el cual se utilizan parámetros de tipo en el tipo de valor de retorno y en la lista de parámetros (figura 18.5). La aplicación utiliza un método genérico llamado `maximo` para determinar y devolver el mayor de sus tres argumentos del mismo tipo. Por desgracia, no se puede utilizar el operador relacional `>` con los tipos de referencia. Sin embargo, es posible comparar dos objetos de la misma clase, si esa clase implementa a la interfaz genérica Comparable< T > (paquete `java.lang`). Todas las clases de envoltura de tipos para los tipos primitivos implementan a esta interfaz. Al igual que las clases genéricas, las **interfaces genéricas** permiten a los programadores especificar, mediante la declaración de una sola interfaz, un conjunto de tipos relacionados. Los objetos Comparable< T > tienen un método llamado `compareTo`. Por ejemplo, si tenemos dos objetos `Integer` llamados `entero1` y `entero2`, éstos pueden compararse con la siguiente expresión:

```
entero1.compareTo( entero2 );
```

Es responsabilidad del programador encargado declarar una clase que implemente a Comparable< T > declarar el método `compareTo`, de tal forma que compare el contenido de dos objetos de esa clase y que devuelva los resultados de la comparación. El método debe devolver `0` si los objetos son iguales, `-1` si `objeto1` es menor que `objeto2`

o `1` si `objeto1` es mayor que `objeto2`. Por ejemplo, el método `compareTo` de la clase `Integer` compara los valores `int` almacenados en dos objetos `Integer`. Un beneficio de implementar la interfaz `Comparable< T >` es que pueden utilizarse objetos `Comparable< T >` con los métodos de ordenamiento y búsqueda de la clase `Collections` (paquete `java.util`). En el capítulo 19, Colecciones, hablaremos sobre esos métodos. En este ejemplo, utilizaremos el método `compareTo` en el método `maximo` para ayudar a determinar el valor más grande.

El método genérico `maximo` (líneas 7 a 18) utiliza el parámetro `T` como el tipo de valor de retorno del método (línea 7), como el tipo de los parámetros `x`, `y` y `z` (línea 7) del método, y como el tipo de la variable local `max` (línea 9). La sección de parámetros de tipo especifica que `T` extiende a `Comparable< T >` (sólo pueden utilizarse objetos de clases que implementen a la interfaz `Comparable< T >` con este método). En este caso, `Comparable` se conoce como el **límite superior** del parámetro de tipo. De manera predeterminada, `Object` es el límite superior. Observe que las declaraciones de los parámetros de tipo que **delimitan** el parámetro siempre utilizan la palabra clave `extends`, sin importar que el parámetro de tipo extienda a una clase o implemente a una interfaz. Este parámetro de tipo es más restrictivo que el que se especifica para `imprimirArreglo` en la figura 18.3, el cual puede imprimir arreglos que contengan cualquier tipo de objeto. La restricción de usar objetos `Comparable < T >` es importante, ya que no todos los objetos se pueden comparar. Sin embargo, se garantiza que los objetos `Comparable< T >` tienen un método `compareTo`.

El método `maximo` utiliza el mismo algoritmo que utilizamos en la sección 6.4 para determinar el mayor de sus tres argumentos. Este método asume que su primer argumento (`x`) es el mayor, y lo asigna a la variable local `max` (línea 9). A continuación, la instrucción `if` en las líneas 11 y 12 determina si `y` es mayor que `max`. La condición invoca al método `compareTo` de `y` con la expresión `y.compareTo( max )`, que devuelve `-1`, `0` o `1`, para determinar la relación de `y` con `max`. Si el valor de retorno de `compareTo` es mayor que `0`, entonces `y` es mayor y se asigna a la variable `max`. De manera similar, la instrucción `if` en las líneas 14 y 15 determina si `z` es mayor que `max`. Si es así, en la línea 15 se asigna `z` a `max`. Después. En la línea 17 se devuelve `max` al método que hizo la llamada.

```
1  // Fig. 18.5: PruebaMaximo.java
2  // El método genérico maximo devuelve el mayor de tres objetos.
3
4  public class PruebaMaximo
5  {
6     // determina el mayor de tres objetos Comparable
7     public static < T extends Comparable< T > > T maximo( T x, T y, T z )
8     {
9        T max = x; // asume que x es el mayor, en un principio
10
11       if ( y.compareTo( max ) > 0 )
12          max = y; // y es el mayor hasta ahora
13
14       if ( z.compareTo( max ) > 0 )
15          max = z; // z es el mayor
16
17       return max; // devuelve el objeto más grande
18    } // fin del método maximo
19
20    public static void main( String args[] )
21    {
22       System.out.printf( "Maximo de %d, %d y %d es %d\n\n", 3, 4, 5,
23          maximo( 3, 4, 5 ) );
24       System.out.printf( "Maximo de %.1f, %.1f y %.1f es %.1f\n\n",
25          6.6, 8.8, 7.7, maximo( 6.6, 8.8, 7.7 ) );
26       System.out.printf( "Maximo de %s, %s y %s es %s\n", "pera",
27          "manzana", "naranja", maximo( "pera", "manzana", "naranja" ) );
28    } // fin de main
29 } // fin de la clase PruebaMaximo
```

Figura 18.5 | El método genérico `maximo`, con un límite superior en su parámetro de tipo. (Parte 1 de 2).

```
Maximo de 3, 4 y 5 es 5

Maximo de 6.6, 8.8 y 7.7 es 8.8

Maximo de pera, manzana y naranja es pera
```

Figura 18.5 | El método genérico `maximo`, con un límite superior en su parámetro de tipo. (Parte 2 de 2).

En `main` (líneas 20 a 28), en la línea 23 se hace una llamada a `maximo` con los enteros `3`, `4` y `5`. Cuando el compilador encuentra esta llamada, primero busca un método `maximo` que reciba tres argumentos de tipo `int`. No hay un método así, por lo cual el compilador busca un método genérico que pueda utilizarse, y encuentra el método genérico `maximo`. Sin embargo, recuerde que los argumentos para un método genérico deben ser de un tipo de referencia. Por lo tanto, el compilador realiza la conversión autoboxing de los tres valores `int` en objetos `Integer`, y especifica que estos tres objetos `Integer` se pasen a `maximo`. Observe que la clase `Integer` (paquete `java.lang`) implementa a la interfaz `Comparable< Integer >` de tal forma que el método `compareTo` compara los valores `int` en dos objetos `Integer`. Por lo tanto, los objetos `Integer` son argumentos válidos para el método `maximo`. Cuando se devuelve el objeto `Integer` que representa el máximo, tratamos de mostrarlo en pantalla con el especificador de formato `%d`, el cual muestra un valor del tipo primitivo `int`. Así, el valor de retorno de `maximo` se muestra en pantalla como un valor `int`.

Hay un proceso similar que ocurre para los tres argumentos `double` que se pasan a `maximo` en la línea 25. Se realiza una conversión autoboxing en cada valor `double` para convertirlo en un objeto `Double` y pasarlo a `maximo`. De nuevo, esto se permite ya que la clase `Double` (paquete `java.lang`) implementa a la interfaz `Comparable< Double >`. El objeto `Double` devuelto por `maximo` se imprime en pantalla con el especificador de formato `%.1f`, el cual muestra un valor del tipo primitivo `double`. Así, se realiza una conversión autounboxing en el valor de retorno de `maximo` y se muestra en pantalla como un `double`. La llamada a `maximo` en la línea 27 recibe tres objetos `String`, que también son objetos `Comparable< String >`. Observe que colocamos de manera intencional el valor más grande en una posición distinta en cada llamada al método (líneas 23, 25 y 27), para mostrar que el método genérico siempre encuentra el valor máximo, sin importar su posición en la lista de argumentos.

Cuando el compilador traduce el método genérico `maximo` en códigos byte de Java, utiliza el borrado (presentado en la sección 18.3) para reemplazar los parámetros de tipo con tipos reales. En la figura 18.3, todos los tipos genéricos se reemplazaron con el tipo `Object`. En realidad, todos los parámetros de tipo se reemplazan con el límite superior del parámetro de tipo; a menos que se especifique lo contrario, `Object` es el límite superior predeterminado. El límite superior de un parámetro de tipo se especifica en la sección de parámetros de tipo. Para indicar el límite superior, coloque después del nombre del parámetro de tipo la palabra clave `extends` y el nombre de la clase o interfaz que representa el límite superior. En la sección de parámetros de tipo del método `maximo` (figura 18.5), especificamos el límite superior como el tipo `Comparable< T >`. Por ende, sólo pueden pasarse objetos `Comparable< T >` como argumentos para `maximo`; cualquier cosa que no sea `Comparable< T >` producirá errores de compilación. La figura 18.6 simula el borrado de los tipos del método máximo, al mostrar el código fuente del método después de eliminar la sección de parámetros de tipo, y después de que se reemplaza

```
public static Comparable maximo(Comparable x, Comparable y, Comparable z)
{
   Comparable max = x; // suponga que al principio x es el más grande

   if ( y.compareTo( max ) > 0 )
      max = y; // y es el mayor hasta ahora

   if ( z.compareTo( max ) > 0 )
      max = z; // z es el mayor

   return max; // devuelve el objeto más grande
} // fin del método maximo
```

Figura 18.6 | El método genérico `maximo`, después de que el compilador realiza el borrado.

el parámetro T con el límite superior, Comparable, en toda la declaración del método. Observe que el borrado de Comparable< T > es simplemente Comparable.

Después del borrado, la versión compilada del método maximo especifica que devuelve el tipo Comparable. Sin embargo, el método que hace la llamada no espera recibir un objeto Comparable. En vez de ello, espera recibir un objeto del mismo tipo que se pasó a maximo como argumento: Integer, Double o String en este ejemplo. Cuando el compilador reemplaza la información del parámetro de tipo con el tipo del límite superior en la declaración del método, también inserta operaciones de conversión explícitas en frente de cada llamada al método, para asegurar que el valor devuelto sea del tipo esperado por el método que hizo la llamada. Así, la llamada a maximo en la línea 23 (figura 18.5) va antecedida por una conversión a Integer, como en

```
(Integer) maximo( 3, 4, 5 )
```

la llamada a maximo en la línea 25 va antecedida por una conversión a Double, como en

```
(Double) maximo( 6.6, 8.8, 7.7 )
```

y la llamada a maximo en la línea 27 va antecedida por una conversión a String, como en

```
(String) maximo( "pera", "manzana", "naranja" )
```

En cada caso, el tipo de la conversión para el valor de retorno se infiere de los tipos de los argumentos del método en cada una de las llamadas al mismo, pues de acuerdo a la declaración del método, el tipo de valor de retorno y los tipos de los argumentos coinciden.

En este ejemplo no podemos usar un método que acepte objetos Object, ya que la clase Object sólo cuenta con una comparación de igualdad. Además, sin los genéricos nosotros seríamos responsables de implementar la operación de conversión. El uso de genéricos asegura que la conversión insertada nunca lance una excepción ClassCastException, asumiendo que utilizamos genéricos en nuestro código (es decir, no debemos mezclar el código anterior con el nuevo código de genéricos).

18.5 Sobrecarga de métodos genéricos

Un método genérico puede sobrecargarse. Una clase puede proporcionar dos o más métodos genéricos que especifiquen el mismo nombre del método, pero distintos parámetros. Por ejemplo, el método genérico imprimirArreglo de la figura 18.3 podría sobrecargarse con otro método genérico imprimirArreglo con los parámetros adicionales subindiceInferior y subindiceSuperior, para especificar la parte del arreglo a imprimir (vea el ejercicio 18.5).

Un método genérico también puede sobrecargarse mediante métodos no genéricos que tengan el mismo nombre del método y el mismo número de parámetros. Cuando el compilador encuentra una llamada al método, busca la declaración del método que coincida con más precisión con el nombre del método y los tipos de los argumentos especificados en la llamada. Por ejemplo, el método genérico imprimirArreglo de la figura 18.3 podría sobrecargarse con una versión específica para objetos String, que imprima estos objetos en un impecable formato tabular (vea el ejercicio 18.6).

Cuando el compilador encuentra una llamada al método, realiza un proceso de asociación para determinar cuál método debe invocar. El compilador trata de encontrar y utilizar una coincidencia precisa, en la que los nombres y tipos de los argumentos de la llamada al método coincidan con los de una declaración específica de ese método. Si no hay un método así, el compilador determina si hay un método inexacto que coincida, y que pueda aplicarse.

18.6 Clases genéricas

El concepto de una estructura de datos, como una pila, puede comprenderse en forma independiente del tipo de elemento que manipula. Las clases genéricas proporcionan los medios para describir el concepto de una pila (o cualquier otra clase) en forma independiente de su tipo. Así, podemos crear instancias de objetos con tipos específicos de la clase genérica. Esta capacidad ofrece una maravillosa oportunidad para la reutilización de software.

Una vez que tenemos una clase genérica, podemos usar una notación concisa para indicar el (los) tipo(s) actual(es) que debe(n) usarse en lugar del (los) parámetro(s) de tipo de la clase. En tiempo de compilación, el compilador de Java asegura la seguridad de los tipos de nuestro código, y utiliza las técnicas de borrado descritas en las secciones 18.3 y 18.4 para permitir que nuestro código cliente interactúe con la clase genérica.

Por ejemplo, una clase `Pila` genérica podría ser la base para crear muchas clases de `Pila` (como, "`Pila` de `Double`", "`Pila` de `Integer`", "`Pila` de `Character`", "`Pila` de `Empleado`"). Estas clases se conocen como **clases parametrizadas** o **tipos parametrizados**, ya que aceptan uno o más parámetros. Recuerde que los parámetros de tipo sólo representan a los tipos de referencias, lo cual significa que la clase genérica `Pila` no puede instanciarse con tipos primitivos. Sin embargo, podemos instanciar una `Pila` que almacene objetos de las clases de envoltura de tipos de Java, y permitir que Java utilice la conversión autoboxing para convertir los valores primitivos en objetos. La conversión autoboxing ocurre cuando un valor de un tipo primitivo (por ejemplo, `int`) se mete en una `Pila` que contiene objetos de clases de envoltura (como, `Integer`). La conversión autounboxing ocurre cuando un objeto de la clase de envoltura se saca de la `Pila` y se asigna a una variable de tipo primitivo.

En la figura 18.7 se presenta una declaración de la clase genérica `Pila`. La declaración de una clase genérica es similar a la de una clase no genérica, excepto que el nombre de la clase va seguido de una sección de parámetros de tipo (línea 4). En este caso, el parámetro de tipo `E` el tipo del elemento que manipulará la `Pila`. Al igual que con los métodos genéricos, la sección de parámetros de tipo de una clase genérica puede tener uno o más parámetros separados por comas. (Usted creará una clase genérica con dos parámetros de tipo en el ejercicio 18.8). El parámetro de tipo `E` se utiliza en la declaración de la clase `Pila` para representar el tipo del elemento. [*Nota:* este ejemplo implementa una `Pila` como un arreglo].

```
1  // Fig. 18.7: Pila.java
2  // La clase genérica Pila.
3
4  public class Pila< E >
5  {
6     private final int tamanio; // número de elementos en la pila
7     private int superior; // ubicación del elemento superior
8     private E[] elementos; // arreglo que almacena los elementos de la pila
9
10    // el constructor sin argumentos crea una pila del tamaño predeterminado
11    public Pila()
12    {
13       this( 10 ); // tamaño predeterminado de la pila
14    } // fin del constructor de Pila sin argumentos
15
16    // constructor que crea una pila del número especificado de elementos
17    public Pila( int s )
18    {
19       tamanio = s > 0 ? s : 10; // establece el tamaño de la Pila
20       superior = -1; // al principio, la Pila está vacía
21
22       elementos = ( E[] ) new Object[ tamanio ]; // crea el arreglo
23    } // fin del constructor de Pila sin argumentos
24
25    // mete un elemento a la pila; si tiene éxito, devuelve verdadero;
26    // en caso contrario, lanza excepción ExcepcionPilaLlena
27    public void push( E valorAMeter )
28    {
29       if ( superior == tamanio - 1 ) // si la pila está llena
30          throw new ExcepcionPilaLlena( String.format(
31                "La Pila esta llena, no se puede meter %s", valorAMeter ) );
32
33       elementos[ ++superior ] = valorAMeter; // coloca valorAMeter en la Pila
34    } // fin del método push
35
36    // devuelve el elemento superior si no está vacía; de lo contrario lanza
      ExcepcionPilaVacia
37    public E pop()
```

Figura 18.7 | Declaración de la clase genérica `Pila`. (Parte 1 de 2).

```
38     {
39        if ( superior == -1 ) // si la pila está vacía
40           throw new ExcepcionPilaVacia( "La Pila esta vacia, no se puede sacar" );
41
42        return elementos[ superior-- ]; // elimina y devuelve el elemento superior de la Pila
43     } // fin del método pop
44  } // fin de la clase Pila< E >
```

Figura 18.7 | Declaración de la clase genérica `Pila`. (Parte 2 de 2).

La clase `Pila` declara la variable `elementos` como un arreglo de tipo `E` (línea 8). Este arreglo almacenará los elementos de la `Pila`. Nos gustaría crear un arreglo de tipo `E` para almacenar los elementos. Sin embargo, el mecanismo de los genéricos no permite parámetros de tipo en las expresiones para crear arreglos, debido a que el parámetro de tipo (en este caso, `E`) no está disponible en tiempo de ejecución. Para crear un arreglo con el tipo apropiado, en la línea 22 del constructor sin argumentos se crea el arreglo como un arreglo de tipo `Object` y se convierte la referencia devuelta por `new` al tipo `E[]`. Cualquier objeto podría almacenarse en un arreglo `Object`, pero el mecanismo de comprobación de tipos del compilador asegura que sólo puedan asignarse al arreglo objetos del tipo declarado de la variable arreglo, a través de cualquier expresión de acceso a arreglos que utilice la variable `elementos`. Aun así, cuando se compila esta clase usando la opción `–Xlint:unchecked`, por ejemplo:

```
javac –Xlint:unchecked Pila.java
```

el compilador emite el siguiente mensaje de advertencia acerca de la línea 22:

```
Stack.java:22: warning: [unchecked] unchecked cast
found   : java.langObject[]
required : E[]
      elementos = ( E[] ) new Object[ tamanio ]; // crea el arreglo
```

La razón de este mensaje es que el compilador no puede asegurar con un 100% de certeza que un arreglo de tipo `Object` nunca contendrá objetos de tipos que no sean `E`. Suponga que `E` representa el tipo `Integer`, de manera que los elementos del arreglo deben almacenar objetos `Integer`. Es posible asignar la variable `elementos` a una variable de tipo `Object[]`, como en

```
Object[] arregloObjetos = elementos;
```

Entonces, cualquier objeto puede colocarse en el arreglo con una instrucción de asignación tal como

```
arregloObjetos[ 0 ] = "hola";
```

Esto coloca un objeto `String` en un arreglo que debe contener sólo objetos `Integer`, lo cual produciría problemas en tiempo de compilación al manejar la `Pila`. Mientras que no ejecute instrucciones como las que se muestran aquí, su `Pila` sólo contendrá objetos del tipo de elemento correcto.

El método `push` (líneas 27 a 34) determina primero si hay un intento de meter un elemento en una `Pila` llena. De ser así, en las líneas 30 y 31 se lanza una `ExcepcionPilaLlena`. La clase `ExcepcionPilaLlena` se declara en la figura 18.8. Si la `Pila` no está llena, en la línea 33 se incrementa el contador `superior` y se coloca el argumento en esa ubicación del arreglo `elementos`.

El método `pop` (líneas 37 a 43) determina primero si hay un intento de sacar un elemento de una `Pila` vacía. De ser así, en la línea 40 se lanza una `ExcepcionPilaVacia`. La clase `ExcepcionPilaVacia` se declara en la figura 18.9. En caso contrario, en la línea 42 se devuelve el elemento superior de la `Pila`, y después se postdecrementa el contador `superior` para indicar la posición del siguiente elemento superior.

Cada una de las clases `ExcepcionPilaLlena` (figura 18.8) y `ExcepcionPilaVacia` (figura 18.9) proporciona el constructor sin argumentos convencional, y un constructor con un argumento de clases de excepciones (como vimos en la sección 13.11). El constructor sin argumentos establece el mensaje de error predeterminado, y el constructor con un argumento establece un mensaje de excepción personalizado.

Al igual que con los métodos genéricos, cuando se compila una clase genérica, el compilador realiza el borrado en los parámetros de tipo de la clase y los reemplaza con sus límites superiores. Para la clase `Pila` (figura 18.7)

no se especifica un límite superior, por lo que se utiliza el límite superior predeterminado, `Object`. El alcance de un parámetro de tipo de una clase genérica es toda la clase. Sin embargo, los parámetros de tipo no se pueden utilizar en las declaraciones `static` de una clase.

```
// Fig. 18.8: ExcepcionPilaLlena.java
// Indica que una pila está llena.
public class ExcepcionPilaLlena extends RuntimeException
{
   // constructor sin argumentos
   public ExcepcionPilaLlena()
   {
      this( "La Pila esta llena" );
   } // fin del constructor de ExcepcionPilaLlena sin argumentos

   // constructor con un argumento
   public ExcepcionPilaLlena( String excepcion )
   {
      super( excepcion );
   } // fin del constructor de ExcepcionPilaLlena sin argumentos
} // fin de la clase ExcepcionPilaLlena
```

Figura 18.8 | Declaración de la clase `ExcepcionPilaLlena`.

```
// Fig. 18.9: ExcepcionPilaVacia.java
// Indica que una pila está llena.
public class ExcepcionPilaVacia extends RuntimeException
{
   // constructor sin argumentos
   public ExcepcionPilaVacia()
   {
      this( "La Pila esta vacia" );
   } // fin del constructor de ExcepcionPilaVacia sin argumentos

   // constructor con un argumento
   public ExcepcionPilaVacia( String excepcion )
   {
      super( excepcion );
   } // fin del constructor de ExcepcionPilaVacia con un argumento
} // fin de la clase ExcepcionPilaVacia
```

Figura 18.9 | Declaración de la clase `ExcepcionPilaVacia`.

Ahora consideremos la aplicación de prueba (figura 18.10) que utiliza la clase genérica `Pila`. En las líneas 9 y 10 se declaran variables de tipo `Pila< Double >` (lo cual se pronuncia como "`Pila` de `Double`") y `Pila< Integer >` (que se pronuncia como "`Pila` de `Integer`"). Los tipos `Double` e `Integer` se conocen como los **argumentos de tipo** de la `Pila`. El compilador los utiliza para reemplazar los parámetros de tipo, de manera que pueda realizar la comprobación de tipos e insertar operaciones de conversión según sea necesario. En breve hablaremos con más detalle sobre las operaciones de conversión. El método `probarPila` (que se llama desde `main`) instancia objetos `pilaDouble` de tamaño `5` (línea 15) y `pilaInteger` de tamaño `10` (línea 16), y después llama a los métodos `pruebaPushDouble` (líneas 25 a 44), `pruebaPopDouble` (líneas 47 a 67), `pruebaPushInteger` (líneas 70 a 89) y `pruebaPopInteger` (líneas 92 a 112), para demostrar las dos `Pilas` en este ejemplo.

El método `pruebaPushDouble` (líneas 25 a 44) invoca al método `push` para colocar en `pilaDouble` los valores `double` 1.1, 2.2, 3.3, 4.4 y 5.5 que se almacenan en el arreglo `elementosDouble`. El ciclo `for` termina cuando el programa de prueba trata de meter un sexto valor en `pilaDouble` (que está llena, ya que `pilaDouble` sólo puede almacenar cinco elementos). En este caso, el método lanza una `ExcepcionPilaLlena` (figura 18.8)

para indicar que la `Pila` está llena. En las líneas 39 a 43 se atrapa esta excepción y se imprime la información de rastreo de la pila. Esta información indica la excepción que ocurrió y muestra que el método `push` de `Pila` generó la excepción en las líneas 30 y 31 del archivo `Pila.java` (figura 18.7). El rastreo también muestra que el método `pruebaPushDouble` de `PruebaPila` llamó al método `push` en la línea 36 de `PruebaPila.java`, que el método `pruebaPilas` llamó al método `probarPushDouble` en la línea 18 de `PruebaPila.java` y que el método `main` llamó al método `pruebaPilas` en la línea 117 de `PruebaPila.java`. Esta información nos permite determinar los métodos que se encontraban en la pila de llamadas a métodos cuando ocurrió la excepción. Debido a que el programa atrapa a la excepción, el entorno en tiempo de ejecución de Java considera que ha sido manejada la excepción y el programa puede continuar su ejecución. Observe que la conversión autoboxing ocurre en la línea 36, cuando el programa trata de meter un valor primitivo `double` en la `pilaDouble`, la cual sólo almacena objetos `Double`.

```
1  // Fig. 18.10: PruebaPila.java
2  // Programa de prueba de la clase genérica Pila.
3
4  public class PruebaPila
5  {
6     private double[] elementosDouble = { 1.1, 2.2, 3.3, 4.4, 5.5, 6.6 };
7     private int[] elementosInteger = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11 };
8
9     private Pila< Double > pilaDouble; // pila que almacena objetos Double
10    private Pila< Integer > pilaInteger; // pila que almacena objetos Integer
11
12    // prueba objetos Pila
13    public void pruebaPilas()
14    {
15       pilaDouble = new Pila< Double >( 5 ); // Pila de objetos Double
16       pilaInteger = new Pila< Integer >( 10 ); // Pila de objetos Integer
17
18       pruebaPushDouble(); // mete valor double en pilaDouble
19       pruebaPopDouble(); // saca de pilaDouble
20       pruebaPushInteger(); // mete valor int en pilaInteger
21       pruebaPopInteger(); // saca de pilaInteger
22    } // fin del método probarPilas
23
24    // prueba el método push con la pila de valores double
25    public void pruebaPushDouble()
26    {
27       // mete elementos en la pila
28       try
29       {
30          System.out.println( "\nMetiendo elementos en pilaDouble" );
31
32          // mete elementos en la Pila
33          for ( double elemento : elementosDouble )
34          {
35             System.out.printf( "%.1f ", elemento );
36             pilaDouble.push( elemento ); // mete en pilaDouble
37          } // fin de for
38       } // fin de try
39       catch ( ExcepcionPilaLlena excepcionPilaLlena )
40       {
41          System.err.println();
42          excepcionPilaLlena.printStackTrace();
43       } // find de catch ExcepcionPilaLlena
44    } // fin del método pruebaPushDouble
```

Figura 18.10 | Programa de prueba de la clase genérica `Pila`. (Parte 1 de 3).

```

    // prueba el método pop con una pila de valores double
    public void pruebaPopDouble()
    {
       // saca elementos de la pila
       try
       {
          System.out.println( "\nSacando elementos de pilaDouble" );
          double valorASacar; // almacena el elemento que se eliminó de la pila

          // elimina todos los elementos de la Pila
          while ( true )
          {
             valorASacar = pilaDouble.pop(); // saca de pilaDouble
             System.out.printf( "%.1f ", valorASacar );
          } // fin de while
       } // fin de try
       catch( ExcepcionPilaVacia excepcionPilaVacia )
       {
          System.err.println();
          excepcionPilaVacia.printStackTrace();
       } // fin de catch ExcepcionPilaVacia
    } // fin del método pruebaPopDouble

    // prueba el método push con pila de valores enteros
    public void pruebaPushInteger()
    {
       // mete elementos a la pila
       try
       {
          System.out.println( "\nMetiendo elementos a pilaInteger" );

          // mete elementos a la Pila
          for ( int elemento : elementosInteger )
          {
             System.out.printf( "%d ", elemento );
             pilaInteger.push( elemento ); // mete a pilaInteger
          } // fin de for
       } // fin de try
       catch ( ExcepcionPilaLlena excepcionPilaLlena )
       {
          System.err.println();
          excepcionPilaLlena.printStackTrace();
       } // fin de catch ExcepcionPilaLlena
    } // fin del método pruebaPushInteger

    // prueba el método pop con una pila de enteros
    public void pruebaPopInteger()
    {
       // saca elementos de la pila
       try
       {
          System.out.println( "\nSacando elementos de pilaInteger" );
          int valorASacar; // almacena el elemento que se eliminó de la pila

          // elimina todos los elementos de la Pila
          while ( true )
          {
             valorASacar = pilaInteger.pop(); // saca de pilaInteger
```

Figura 18.10 | Programa de prueba de la clase genérica `Pila`. (Parte 2 de 3).

```
104               System.out.printf( "%d ", valorASacar );
105            } // fin de while
106         } // fin de try
107         catch( ExcepcionPilaVacia excepcionPilaVacia )
108         {
109            System.err.println();
110            excepcionPilaVacia.printStackTrace();
111         } // fin de catch ExcepcionPilaVacia
112      } // fin del método pruebaPopInteger
113
114      public static void main( String args[] )
115      {
116         PruebaPila aplicacion = new PruebaPila();
117         aplicacion.probarPilas();
118      } // fin de main
119   } // fin de la clase PruebaPila
```

```
Metiendo elementos en pilaDouble
1.1 2.2 3.3 4.4 5.5 6.6
ExcepcionPilaLlena: La Pila esta llena, no se puede meter 6.6
        at Pila.push(Pila.java:30)
        at PruebaPila.pruebaPushDouble(PruebaPila.java:36)
        at PruebaPila.probarPilas(PruebaPila.java:18)
        at PruebaPila.main(PruebaPila.java:117)

Sacando elementos de pilaDouble
5.5 4.4 3.3 2.2 1.1
ExcepcionPilaVacia: La Pila esta vacia, no se puede sacar
        at Pila.pop(Pila.java:40)
        at PruebaPila.pruebaPopDouble(PruebaPila.java:58)
        at PruebaPila.probarPilas(PruebaPila.java:19)
        at PruebaPila.main(PruebaPila.java:117)

Metiendo elementos a pilaInteger
1 2 3 4 5 6 7 8 9 10 11
ExcepcionPilaLlena: La Pila esta llena, no se puede meter 11
        at Pila.push(Pila.java:30)
        at PruebaPila.pruebaPushInteger(PruebaPila.java:81)
        at PruebaPila.probarPilas(PruebaPila.java:20)
        at PruebaPila.main(PruebaPila.java:117)

Sacando elementos de pilaInteger
10 9 8 7 6 5 4 3 2 1
ExcepcionPilaVacia: La Pila esta vacia, no se puede sacar
        at Pila.pop(Pila.java:40)
        at PruebaPila.pruebaPopInteger(PruebaPila.java:103)
        at PruebaPila.probarPilas(PruebaPila.java:21)
        at PruebaPila.main(PruebaPila.java:117)
```

Figura 18.10 | Programa de prueba de la clase genérica `Pila`. (Parte 3 de 3).

El método `pruebaPopDouble` (líneas 47 a 67) invoca al método `pop` de `Pila` en un ciclo `while` infinito para eliminar todos los valores de la pila. Observe en los resultados que los valores se sacan sin duda en el orden último en entrar, primero en salir (desde luego que ésta es la característica que define a las pilas). El ciclo `while` (líneas 57 a 61) continúa hasta que la pila está vacía (es decir, hasta que ocurre una `ExcepcionPilaVacia`), lo cual hace que el programa continúe con el bloque `catch` (líneas 62 a 66) y maneje la excepción, para que pueda continuar su ejecución. Cuando el programa de prueba trata de sacar un sexto valor, la `pilaDouble` está vacía, por lo que el método `pop` lanza una `ExcepcionPilaVacia`. La conversión autoboxing ocurre en la línea 58, en donde el programa asigna el objeto `Double` que se sacó de la pila a una variable primitiva `double`. En la sección 18.4 vimos

que el compilador inserta operaciones de conversión para asegurar que se devuelvan los tipos apropiados de los métodos genéricos. Después del borrado, el método `pop` de `Pila` devuelve el tipo `Object`. Sin embargo, el código cliente en el método `pruebaPopDouble` espera recibir un valor `double` cuando regresa el método `pop`. Así, el compilador inserta una conversión a `Double`, como en

```
valorASacar = ( Double ) pilaDouble.pop();
```

para asegurar que se devuelva una referencia del tipo apropiado, que se realice la conversión autounboxing y se asigne a `valorASacar`.

El método `pruebaPushInteger` (líneas 70 a 89) invoca el método `push` de `Pila` para colocar valores en `PruebaInteger` hasta que esté llena. El método `pruebaPopInteger` (líneas 92 a 112) invoca el método `pop` de `Prueba` para eliminar valores de `pilaInteger` hasta que esté vacía. Una vez más, observe que los valores se sacan en el orden último en entrar, primero en salir. Durante el proceso de borrado, el compilador reconoce que el código cliente en el método `pruebaPopInteger` espera recibir un valor `int` cuando regresa el método `pop`. Por lo tanto, el compilador inserta una conversión a `Integer`, como en

```
valorASacar = ( Integer ) pilaInteger.pop();
```

para asegurar que se devuelva una referencia del tipo apropiado, se realice una conversión autounboxing y se asigne a `valorASacar`.

Creación de métodos genéricos para probar la clase `Pila< E >`

Observe que el código en los métodos `pruebaPushDouble` y `pruebaPushInteger` es casi idéntico para meter valores en una `Pila<Double>` o una `Pila<Integer>`, respectivamente, y que el código en los métodos `pruebaPopDouble` y `pruebaPopInteger` es casi idéntico para sacar valores de una `Pila<Double>` o una `Pila<Integer>`, respectivamente. Esto presenta otra oportunidad para utilizar los métodos genéricos. En la figura 18.11 se declara el método genérico `probarPush` (líneas 26 a 46) para que realice las mismas tareas que `pruebaPushDouble` y `pruebaPushInteger` en la figura 18.10; es decir, meter valores en una `Pila< T >`. De manera similar, el método genérico `pruebaPop` (líneas 49 a 69) realiza las mismas tareas que `pruebaPopDouble` y `pruebaPopInteger` en la figura 18.10; es decir, sacar valores de una `Pila< T >`. Observe que la salida de la figura 18.11 coincide precisamente con la salida de la figura 18.10.

El método `probarPilas` (líneas 14 a 23) crea los objetos `Pila< Double >` (línea 16) y la `Pila< Integer >` (línea 17). En las líneas 19 a 22 se invocan los métodos genéricos `pruebaPush` y `pruebaPop` para probar los objetos `Pila`. Recuerde que los parámetros de tipo sólo pueden representar tipos de referencias. Por lo tanto, para poder pasar los arreglos `elementosDouble` y `elementosInteger` al método genérico `pruebaPush`, los arreglos declarados en las líneas 6 a 8 deben declararse con los tipos de envoltura `Double` e `Integer`. Cuando estos arreglos se inicializan con valores primitivos, el compilador realiza conversiones autoboxing en cada valor primitivo.

```
1  // Fig. 18.11: PruebaPila2.java
2  // Programa de prueba de la clase genérica Pila.
3
4  public class PruebaPila2
5  {
6     private Double[] elementosDouble = { 1.1, 2.2, 3.3, 4.4, 5.5, 6.6 };
7     private Integer[] elementosInteger =
8        { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11 };
9
10    private Pila< Double > pilaDouble; // pila que almacena objetos Double
11    private Pila< Integer > pilaInteger; // pila que almacena objetos Integer
12
13    // prueba los objetos Pila
14    public void probarPilas()
15    {
16       pilaDouble = new Pila< Double >( 5 ); // Pila de objetos Double
17       pilaInteger = new Pila< Integer >( 10 ); // Pila de objetos Integer
```

Figura 18.11 | Paso de una `Pila` de tipo genérico a un método genérico. (Parte 1 de 3).

```

      probarPush( "pilaDouble", pilaDouble, elementosDouble );
      probarPop( "pilaDouble", pilaDouble );
      probarPush( "pilaInteger", pilaInteger, elementosInteger );
      probarPop( "pilaInteger", pilaInteger );
   } // fin del método probarPilas

   // el método genérico probarPush mete elementos en una Pila
   public < T > void probarPush( String nombre, Pila< T > pila,
      T[] elementos )
   {
      // mete elementos a la pila
      try
      {
         System.out.printf( "\nMetiendo elementos a %s\n", nombre );

         // mete elementos a la Pila
         for ( T elemento : elementos )
         {
           System.out.printf( "%s ", elemento );
           pila.push( elemento ); // mete elemento a la pila
         }
      } // fin de try
      catch ( ExcepcionPilaLlena excepcionPilaLlena )
      {
         System.out.println();
         excepcionPilaLlena.printStackTrace();
      } // fin de catch ExcepcionPilaLlena
   } // fin del método probarPush

   // el método genérico probarPop saca elementos de una Pila
   public < T > void probarPop( String nombre, Pila< T > pila )
   {
      // saca elementos de la pila
      try
      {
         System.out.printf( "\nSacando elementos de %s\n", nombre );
         T valorASacar; // almacena el elemento eliminado de la pila

         // elimina todos los elementos de la Pila
         while ( true )
         {
           valorASacar = pila.pop(); // saca de la pila
           System.out.printf( "%s ", valorASacar );
         } // fin de while
      } // fin de try
      catch( ExcepcionPilaVacia excepcionPilaVacia )
      {
         System.out.println();
         excepcionPilaVacia.printStackTrace();
      } // fin de catch ExcepcionPilaVacia
   } // fin del método probarPop

   public static void main( String args[] )
   {
      PruebaPila2 aplicacion = new PruebaPila2();
      aplicacion.probarPilas();
   } // fin de main
} // fin de la clase PruebaPila2
```

Figura 18.11 | Paso de una `Pila` de tipo genérico a un método genérico. (Parte 2 de 3).

```
Metiendo elementos a pilaDouble
1.1 2.2 3.3 4.4 5.5 6.6
ExcepcionPilaLlena: La Pila esta llena, no se puede meter 6.6
       at Pila.push(Pila.java:30)
       at PruebaPila2.probarPush(PruebaPila2.java:38)
       at PruebaPila2.probarPilas(PruebaPila2.java:19)
       at PruebaPila2.main(PruebaPila2.java:74)

Sacando elementos de pilaDouble
5.5 4.4 3.3 2.2 1.1
ExcepcionPilaVacia: La Pila esta vacia, no se puede sacar
       at Pila.pop(Pila.java:40)
       at PruebaPila2.probarPop(PruebaPila2.java:60)
       at PruebaPila2.probarPilas(PruebaPila2.java:20)
       at PruebaPila2.main(PruebaPila2.java:74)

Metiendo elementos a pilaInteger
1 2 3 4 5 6 7 8 9 10 11
ExcepcionPilaLlena: La Pila esta llena, no se puede meter 11
       at Pila.push(Pila.java:30)
       at PruebaPila2.probarPush(PruebaPila2.java:38)
       at PruebaPila2.probarPilas(PruebaPila2.java:21)
       at PruebaPila2.main(PruebaPila2.java:74)

Sacando elementos de pilaInteger
10 9 8 7 6 5 4 3 2 1
ExcepcionPilaVacia: La Pila esta vacia, no se puede sacar
       at Pila.pop(Pila.java:40)
       at PruebaPila2.probarPop(PruebaPila2.java:60)
       at PruebaPila2.probarPilas(PruebaPila2.java:22)
       at PruebaPila2.main(PruebaPila2.java:74)
```

Figura 18.11 | Paso de una `Pila` de tipo genérico a un método genérico. (Parte 3 de 3).

El método genérico `probarPush` (líneas 26 a 46) usa el parámetro de tipo `T` (especificado en la línea 26) para representar el tipo de datos almacenado en la `Pila< T >`. El método genérico recibe tres argumentos: un `String` que representa el nombre del objeto `Pila< T >` para fines de mostrarlo en pantalla, una referencia a un objeto de tipo `Pila< T >` y un arreglo de tipo `T`; el tipo de elementos que se meterán en la `Pila< T >`. Observe que el compilador hace valer la consistencia entre el tipo de la `Pila` y los elementos que se meterán en la misma cuando se invoque a `push`, lo cual es el valor real de la llamada al método genérico. El método genérico `probarPop` (líneas 49 a 69) recibe dos argumentos: un `String` que represente el nombre del objeto `Pila< T >` para fines de mostrarlo en pantalla, y una referencia a un objeto de tipo `Pila< T >`.

18.7 Tipos crudos (raw)

Los programas de prueba para la clase genérica `Pila` en la sección 18.6 crea instancias de objetos `Pila` con los argumentos de tipo `Double` e `Integer`. También es posible instanciar la clase genérica `Pila` sin especificar un argumento de tipo, como se muestra a continuación:

```
Stack pilaObjetos = new Stack( 5 ); // no se especifica un argumento de tipo
```

En este caso, se dice que la `pilaObjetos` tiene un `tipo crudo`, lo cual significa que el compilador utiliza de manera implícita el tipo `Object` en la clase genérica para cada argumento de tipo. Así, la instrucción anterior crea una `Pila` que puede almacenar objetos de cualquier tipo. Esto es importante para la compatibilidad inversa con versiones anteriores de Java. Por ejemplo, todas las estructuras de datos del Marco de trabajo Collections de Java (vea el capítulo 19, Colecciones) almacenan referencias a objetos `Object`, pero ahora se implementan como tipos genéricos.

A una variable `Pila` de tipo crudo se le puede asignar una `Pila` que especifique un argumento de tipo, como un objeto `Pila< Double >`, de la siguiente manera:

```
Pila  pilaTipoCrudo2 = new Pila< Double >( 5 );
```

debido a que el tipo `Double` es una subclase de `Object`. La asignación se permite ya que los elementos en una `Pila< Double >` (es decir, objetos `Double`) son ciertamente objetos; la clase `Double` es una subclase indirecta de `Object`.

De manera similar, a una variable `Pila` que especifica a un argumento de tipo en su declaración se le puede asignar un objeto `Pila` de tipo crudo, como en:

```
Pila< Integer > pilaInteger = new Pila( 10 );
```

Aunque esta asignación está permitida, no es segura debido a que una `Pila` de tipo crudo podría almacenar tipos distintos de `Integer`. En este caso, el compilador genera un mensaje de advertencia, el cual indica la asignación insegura.

El programa de prueba de la figura 18.12 utiliza la noción de un tipo crudo. En la línea 14 se crea una instancia de la clase genérica `Pila` con un tipo crudo, lo cual indica que `pilaTipoCrudo1` puede contener objetos de cualquier tipo. En la línea 17 se asigna una `Pila< Double >` a la variable `pilaTipoCrudo2`, la cual se declara como una `Pila` de tipo crudo. En la línea 20 se asigna una `Pila` de tipo crudo a la variable `Pila< Integer >`, lo cual es legal pero hace que el compilador genere un mensaje de advertencia (figura 18.13), indicando una asignación potencialmente insegura; de nuevo, esto ocurre debido a que una `Pila` de tipo crudo podría almacenar tipos distintos de `Integer`. Además, cada una de las llamadas al método genérico `probarPush` y `probarPop` en las líneas 22 a 25 produce un mensaje de advertencia del compilador (figura 18.13). Estas advertencias ocurren debido a que las variables `pilaTipoCrudo1` y `pilaTipoCrudo2` se declaran como `Pilas` de tipo crudo, pero los métodos `probarPush` y `probarPop` esperan un segundo argumento que sea una `Pila` con un argumento de tipo específico. Las advertencias indican que el compilador no puede garantizar que los tipos manipulados por las pilas sean los correctos, ya que no suministramos una variable declarada con un argumento de tipo. Los métodos `probarPush` (líneas 31 a 51) y `probarPop` (líneas 54 a 74) son iguales que en la figura 18.11.

```
1  // Fig. 18.12: PruebaTipoCrudo.java
2  // Programa de prueba de tipos crudos.
3
4  public class PruebaTipoCrudo
5  {
6     private Double[] elementosDouble = { 1.1, 2.2, 3.3, 4.4, 5.5, 6.6 };
7     private Integer[] elementosInteger =
8        { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11 };
9
10    // método para evaluar Pilas con tipos crudos
11    public void probarPilas()
12    {
13       // Pila de tipos crudos asignada a una variable Pila de tipos crudos
14       Pila pilaTipoCrudo1 = new Pila( 5 );
15
16       // Pila< Double > asignada a una variable Pila de tipos crudos
17       Pila pilaTipoCrudo2 = new Pila< Double >( 5 );
18
19       // Pila de tipos crudos asignada a una variable Pila< Integer >
20       Pila< Integer > pilaInteger = new Pila( 10 );
21
22       probarPush( "pilaTipoCrudo1", pilaTipoCrudo1, elementosDouble );
23       probarPop( "pilaTipoCrudo1", pilaTipoCrudo1 );
24       probarPush( "pilaTipoCrudo2", pilaTipoCrudo2, elementosDouble );
25       probarPop( "pilaTipoCrudo2", pilaTipoCrudo2 );
26       probarPush( "pilaInteger", pilaInteger, elementosInteger );
```

Figura 18.12 | Programa de prueba de tipos crudos. (Parte 1 de 3).

```
      probarPop( "pilaInteger", pilaInteger );
   } // fin del método probarPilas

   // método genérico que mete elementos a la pila
   public < T > void probarPush( String nombre, Pila< T > pila,
      T[] elementos )
   {
      // mete elementos a la pila
      try
      {
         System.out.printf( "\nMetiendo elementos a %s\n", nombre );

         // mete elementos a la Pila
         for ( T elemento : elementos )
         {
            System.out.printf( "%s ", elemento );
            pila.push( elemento ); // mete elemento a la pila
         } // fin de for
      } // fin de try
      catch ( ExcepcionPilaLlena excepcionPilaLlena )
      {
         System.out.println();
         excepcionPilaLlena.printStackTrace();
      } // fin de catch ExcepcionPilaLlena
   } // fin del método probarPush

   // método genérico probarPop para sacar elementos de la pila
   public < T > void probarPop( String nombre, Pila< T > pila )
   {
      // saca elementos de la pila
      try
      {
         System.out.printf( "\nSacando elementos de %s\n", nombre );
         T valorASacar; // almacena el elemento eliminado de la pila

         // elimina elementos de la Pila
         while ( true )
         {
            valorASacar = pila.pop(); // saca de la pila
            System.out.printf( "%s ", valorASacar );
         } // fin de while
      } // fin de try
      catch( ExcepcionPilaVacia excepcionPilaVacia )
      {
         System.out.println();
         excepcionPilaVacia.printStackTrace();
      } // fin de catch ExcepcionPilaVacia
   } // fin del método probarPop

   public static void main( String args[] )
   {
      PruebaTipoCrudo aplicacion = new PruebaTipoCrudo();
      aplicacion.probarPilas();
   } // fin de main
} // fin de la clase PruebaTipoCrudo
```

```
Metiendo elementos a pilaTipoCrudo1
1.1 2.2 3.3 4.4 5.5 6.6
ExcepcionPilaLlena: La Pila esta llena, no se puede meter 6.6
```

Figura 18.12 | Programa de prueba de tipos crudos. (Parte 2 de 3).

```
        at Pila.push(Pila.java:30)
        at PruebaTipoCrudo.probarPush(PruebaTipoCrudo.java:43)
        at PruebaTipoCrudo.probarPilas(PruebaTipoCrudo.java:22)
        at PruebaTipoCrudo.main(PruebaTipoCrudo.java:79)

Sacando elementos de pilaTipoCrudo1
5.5 4.4 3.3 2.2 1.1
ExcepcionPilaVacia: La Pila esta vacia, no se puede sacar
        at Pila.pop(Pila.java:40)
        at PruebaTipoCrudo.probarPop(PruebaTipoCrudo.java:65)
        at PruebaTipoCrudo.probarPilas(PruebaTipoCrudo.java:23)
        at PruebaTipoCrudo.main(PruebaTipoCrudo.java:79)

Metiendo elementos a pilaTipoCrudo2
1.1 2.2 3.3 4.4 5.5 6.6
ExcepcionPilaLlena: La Pila esta llena, no se puede meter 6.6
        at Pila.push(Pila.java:30)
        at PruebaTipoCrudo.probarPush(PruebaTipoCrudo.java:43)
        at PruebaTipoCrudo.probarPilas(PruebaTipoCrudo.java:24)
        at PruebaTipoCrudo.main(PruebaTipoCrudo.java:79)

Sacando elementos de pilaTipoCrudo2
5.5 4.4 3.3 2.2 1.1
ExcepcionPilaVacia: La Pila esta vacia, no se puede sacar
        at Pila.pop(Pila.java:40)
        at PruebaTipoCrudo.probarPop(PruebaTipoCrudo.java:65)
        at PruebaTipoCrudo.probarPilas(PruebaTipoCrudo.java:25)
        at PruebaTipoCrudo.main(PruebaTipoCrudo.java:79)

Metiendo elementos a pilaInteger
1 2 3 4 5 6 7 8 9 10 11
ExcepcionPilaLlena: La Pila esta llena, no se puede meter 11
        at Pila.push(Pila.java:30)
        at PruebaTipoCrudo.probarPush(PruebaTipoCrudo.java:43)
        at PruebaTipoCrudo.probarPilas(PruebaTipoCrudo.java:26)
        at PruebaTipoCrudo.main(PruebaTipoCrudo.java:79)

Sacando elementos de pilaInteger
10 9 8 7 6 5 4 3 2 1
ExcepcionPilaVacia: La Pila esta vacia, no se puede sacar
        at Pila.pop(Pila.java:40)
        at PruebaTipoCrudo.probarPop(PruebaTipoCrudo.java:65)
        at PruebaTipoCrudo.probarPilas(PruebaTipoCrudo.java:27)
        at PruebaTipoCrudo.main(PruebaTipoCrudo.java:79)
```

Figura 18.12 | Programa de prueba de tipos crudos. (Parte 3 de 3).

La figura 18.13 muestra los mensajes de advertencia generados por el compilador (al compilar con la opción -Xlint:unchecked) cuando se compila el archivo PruebaTipoCrudo.java (figura 18.12). La primera advertencia se genera para la línea 20, en la cual se asigna un tipo crudo Pila a una variable Pila< Integer >; el compilador no puede asegurar que todos los objetos en la Pila sean objetos Integer. La segunda advertencia se genera para la línea 22. Debido a que el segundo argumento del método es una variable Pila de tipo crudo, el compilador determina el argumento de tipo para el método probarPush del arreglo Double que se pasa como tercer argumento. En este caso, Double es el argumento de tipo, por lo que el compilador espera que se pase una Pila< Double > como segundo argumento. La advertencia ocurre debido a que el compilador no puede asegurar que una Pila de tipo crudo contenga sólo objetos Double. La advertencia en la línea 24 ocurre por la misma razón, aun cuando la Pila actual a la que pilaTipoCrudo2 hace referencia es una Pila< Double >. El compilador no puede garantizar que la variable siempre hará referencia al mismo objeto Pila, por lo que debe utilizar el tipo declarado de la variable para realizar toda la comprobación de tipos. En las líneas 23 y 25 se

```
PruebaTipoCrudo.java:20: warning: unchecked assignment
found   : Pila
required: Pila<java.lang.Integer>
      Pila< Integer > pilaInteger = new Pila( 10 );
                                    ^
PruebaTipoCrudo.java:22: warning: [unchecked] unchecked method invocation:
<T>probarPush(java.lang.String,Pila<T>,T[]) in PruebaTipoCrudo is applied to
(java.lang.String,Pila,java.lang.Double[])
      probarPush( "pilaTipoCrudo1", pilaTipoCrudo1, elementosDouble );
      ^
PruebaTipoCrudo.java:23: warning: [unchecked] unchecked method invocation:
<T>probarPop(java.lang.String,Pila<T>) in PruebaTipoCrudo is applied to (
java.lang.String,Pila)
      probarPop( "pilaTipoCrudo1", pilaTipoCrudo1 );
      ^
PruebaTipoCrudo.java:24: warning: [unchecked] unchecked method invocation:
<T>probarPush(java.lang.String,Pila<T>,T[]) in PruebaTipoCrudo is applied to
(java.lang.String,Pila,java.lang.Double[])
      probarPush( "pilaTipoCrudo2", pilaTipoCrudo2, elementosDouble );
      ^
PruebaTipoCrudo.java:25: warning: [unchecked] unchecked method invocation:
<T>probarPop(java.lang.String,Pila<T>) in PruebaTipoCrudo is applied to
(java.lang.String,Pila)
      probarPop( "pilaTipoCrudo2", pilaTipoCrudo2 );
      ^
5 warnings
```

Figura 18.13 | Mensajes de advertencia del compilador.

generan advertencias debido a que el método `probarPop` espera como argumento una `Pila` para la cual se haya especificado un argumento de tipo. Sin embargo, en cada llamada a `probarPop`, pasamos una variable `Pila` de tipo crudo. Por ende, el compilador indica una advertencia, debido a que no puede comprobar los tipos utilizados en el cuerpo del método.

18.8 Comodines en métodos que aceptan parámetros de tipo

En esta sección presentaremos un poderoso concepto de los genéricos, conocido como los **comodines**. Para este fin, también introduciremos una nueva estructura de datos del paquete `java.util`. El capítulo 19, Colecciones, habla sobre el Marco de trabajo Collections de Java, el cual proporciona muchas estructuras de datos genéricas y algoritmos que manipulan a los elementos de estas estructuras de datos. Tal vez la más simple de estas estructuras de datos sea la clase `ArrayList`: una estructura de datos tipo arreglo, que puede cambiar su tamaño en forma dinámica. Como parte de esta discusión, aprenderá a crear un objeto `ArrayList`, a añadirle elementos y a recorrer esos elementos mediante el uso de una instrucción `for` mejorada.

Antes de presentar los comodines, analicemos un ejemplo que nos ayude a motivar su uso. Suponga que desea implementar un método genérico llamado `suma`, que obtenga el total de números en una colección, como en un objeto `ArrayList`. Para empezar, podría insertar los números en la colección. Como sabe, las clases genéricas sólo se pueden utilizar con tipos de clases o de interfaces. Por lo tanto, se realizaría una conversión autoboxing de los números a objetos de las clases de envoltura de tipos. Por ejemplo, cualquier valor `int` se convertiría mediante autoboxing en un objeto `Integer`, y cualquier valor `double` se convertiría mediante autoboxing en un objeto `Double`. Nos gustaría poder obtener el total de todos los números en el objeto `ArrayList`, sin importar su tipo. Por esta razón, declaramos el objeto `ArrayList` con el argumento de tipo `Number`, el cual es la superclase tanto de `Integer` como de `Double`. Además, el método `suma` recibirá un parámetro de tipo `ArrayList< Number >` y obtendrá el total de sus elementos. En la figura 18.14 se demuestra cómo obtener el total de los elementos de un objeto `ArrayList` de objetos `Number`.

En la línea 11 se declara e inicializa un arreglo de objetos `Number`. Como los inicializadores son valores primitivos, Java realiza conversiones autoboxing en cada valor primitivo, para convertirlo en un objeto de su correspondiente tipo de envoltura. Los valores `int 1` y `3` se convierten mediante autoboxing en objetos `Integer`,

```
1  // Fig. 18.14: TotalNumeros.java
2  // Suma de los elementos de un objeto ArrayList.
3  import java.util.ArrayList;
4
5  public class TotalNumeros
6  {
7     public static void main( String args[] )
8     {
9        // crea, inicializa y muestra en pantalla el objeto ArrayList de objetos Number
10       // que contiene objetos Integer y Double, después muestra el total de los elementos
11       Number[] numeros = { 1, 2.4, 3, 4.1 }; // objetos Integer y Double
12       ArrayList< Number > listaNumeros = new ArrayList< Number >();
13
14       for ( Number elemento : numeros )
15          listaNumeros.add( elemento ); // coloca cada número en listaNumeros
16
17       System.out.printf( "listaNumeros contiene: %s\n", listaNumeros );
18       System.out.printf( "Total de los elementos en listaNumeros: %.1f\n",
19          sumar( listaNumeros ) );
20    } // fin de main
21
22    // calcula el total de los elementos de ArrayList
23    public static double sumar( ArrayList< Number > lista )
24    {
25       double total = 0; // inicializa el total
26
27       // calcula la suma
28       for ( Number elemento : lista )
29          total += elemento.doubleValue();
30
31       return total;
32    } // fin del método sumar
33 } // fin de la clase TotalNumeros
```

```
listaNumeros contiene: [1, 2.4, 3, 4.1]
Total de los elementos en listaNumeros: 10.5
```

Figura 18.14 | Total de los números en `ArrayList< Number >`.

y los valores `double 2.4` y `4.1` se convierten mediante autoboxing en objetos `Double`. En la línea 12 se declara y crea un objeto `ArrayList` que almacena objetos `Number`, y se asigna a la variable `listaNumeros`. Observe que no tenemos que especificar el tamaño del objeto `ArrayList`, ya que crecerá de manera automática, a medida que insertemos objetos.

En las líneas 14 y 15 se recorre el arreglo `numeros` y se coloca cada uno de sus elementos en `listaNumeros`. El método `add` de la clase `ArrayList` anexa un elemento al final de la colección. En la línea 17 se imprime en pantalla el contenido del objeto `ArrayList` como un objeto `String`. Esta instrucción invoca en forma implícita al método `toString` de `ArrayList`, el cual devuelve una cadena de la forma "[*elementos*]", en la cual *elementos* es una lista separada por comas de las representaciones de cadena de los elementos. En las líneas 18 y 19 se muestra la suma de los elementos que se devuelve mediante la llamada al método suma en la línea 19.

El método `sumar` (líneas 23 a 32) recibe un objeto `ArrayList` de objetos `Number` y calcula el total de los objetos `Number` en la colección. El método utiliza valores `double` para realizar los cálculos y devuelve el resultado como un `double`. En la línea 25 se declara la variable local `total` y se inicializa en `0`. En las líneas 28 y 29 se utiliza la instrucción `for` mejorada, la cual está diseñada para trabajar con arreglos y con las colecciones del Marco de trabajo Collections, para obtener el total de los elementos del objeto `ArrayList`. La instrucción `for` asigna cada objeto `Number` en el objeto `ArrayList` a la variable `elemento`, y después utiliza el método `doubleValue` de la clase `Number` para obtener el valor primitivo subyacente del objeto `Number` como un valor `double`. El resultado se suma a `total`. Cuando el ciclo termina, el método devuelve el total.

Cómo implementar el método suma con un argumento de tipo comodín en su parámetro

Recuerde que el propósito del método `suma` en la figura 18.14 era obtener el total de cualquier tipo de objetos `Number` almacenados en un objeto `ArrayList`. Creamos un objeto `ArrayList` de objetos `Number` que contenía objetos `Integer` y `Double`. Los resultados de la figura 18.14 demuestran que el método `sumar` trabajó correctamente. Dado que el método `sumar` puede obtener el total de los elementos de un objeto `ArrayList` de objetos `Number`, podríamos esperar que el método también funcionara para objetos `ArrayList` que contengan elementos de sólo un tipo numérico, como `ArrayList< Integer >`. Así, modificamos la clase `TotalNumeros` para crear un objeto `ArrayList` de objetos `Integer` y pasarlo al método `sumar`. Al compilar el programa, el compilador genera el siguiente mensaje de error:

```
sumar(java.util.ArrayList<java.lang.Number>) in TotalNumerosErrores
cannot be applied to (java.util.ArrayList<java.lang.Integer>)
```

Aunque `Number` es la superclase de `Integer`, el compilador no considera que el tipo parametrizado `ArrayList< Number >` sea un supertipo de `ArrayList< Integer >`. Si lo fuera, entonces toda operación que pudiéramos realizar en `ArrayList< Number >` funcionaría también en un `ArrayList< Integer >`. Considere el hecho de que puede sumar un objeto `Double` a un `ArrayList< Number >`, debido a que un `Double` *es un* `Number`, pero no se puede sumar un objeto `Double` a un `ArrayList< Integer >`, ya que un `Double` *no es* un `Integer`. Por ende, no es válida la relación de los subtipos.

¿Cómo creamos una versión más flexible del método `sumar` que pueda obtener el total de los elementos de cualquier objeto `ArrayList` que contenga elementos de cualquier subclase de `Number`? Aquí es donde son importantes los **argumentos tipo comodín**. Los comodines nos permiten especificar parámetros de métodos, valores de retorno, variables o campos, etc., que actúan como supertipos de los tipos parametrizados. En la figura 18.15, el parámetro del método `suma` se declara en la línea 50 con el tipo:

```
ArrayList< ? extends Number >
```

Un argumento tipo comodín se denota mediante un signo de interrogación (**?**), que por sí solo representa un “tipo desconocido”. En este caso, el comodín extiende a la clase `Number`, lo cual significa que el comodín tiene un límite superior de `Number`. Por ende, el argumento de tipo desconocido debe ser `Number` o una subclase de `Number`. Con el tipo del parámetro que se muestra aquí, el método `sumar` puede recibir un argumento `ArrayList` que contenga cualquier tipo de `Number`, como `ArrayList< Integer >` (línea 20), `ArrayList< Double >` (línea 33) o `ArrayList< Number >` (línea 46).

En las líneas 11 a 20 se crea e inicializa un objeto `ArrayList< Integer >` llamado `listaEnteros`, se imprimen en pantalla sus elementos y se obtiene el total de los mismos mediante una llamada al método `sumar` (línea 20). En las líneas 24 a 33 se realizan las mismas operaciones para un objeto `ArrayList< Double >` llamado `listaDouble`. En las líneas 37 a 46 se realizan las mismas operaciones para un objeto `ArrayList< Number >` llamado `listaNumeros`, el cual contiene objetos `Integer` y `Double`.

En el método `sumar` (líneas 50 a 59), aunque los tipos de los elementos del argumento `ArrayList` no son directamente conocidos para el método, se sabe que por lo menos son de tipo `Number`, ya que el comodín se especificó con el límite superior `Number`. Por esta razón se permite la línea 56, ya que todos los objetos `Number` tienen un método `doubleValue`.

Aunque los comodines proporcionan una flexibilidad al pasar tipos parametrizados a un método, también tienen ciertas desventajas. Como el comodín (?) en el encabezado del método (línea 50) no especifica el nombre de un parámetro de tipo, no se puede utilizar como nombre de tipo en el cuerpo del método (es decir, no podemos reemplazar `Numero` con `?` en la línea 55). Sin embargo, podríamos declarar el método `sumar` de la siguiente manera:

```
public static <T extends Number> double sumar( ArrayList< T > lista )
```

lo cual permite al método recibir un objeto `ArrayList` que contenga elementos de cualquier subclase de `Number`. Después, podríamos usar el parámetro de tipo `T` en el cuerpo del método.

Si el comodín se especifica sin un límite superior, entonces sólo se pueden invocar los métodos del tipo `Object` en valores del tipo del comodín. Además, los métodos que utilizan comodines en los argumentos de tipo de sus parámetros no pueden utilizarse para agregar elementos a una colección a la que el parámetro hace referencia.



Error común de programación 18.4

Utilizar un comodín en la sección de parámetros de tipo de un método, o utilizar un comodín como un tipo explícito de una variable en el cuerpo del método, es un error de sintaxis.

```java
// Fig. 18.15: PruebaComodin.java
// Programa de prueba de comodines.
import java.util.ArrayList;

public class PruebaComodin
{
   public static void main( String args[] )
   {
      // crea, inicializa e imprime en pantalla un objeto ArrayList de objetos Integer,
      // y después muestra el total de los elementos
      Integer[] enteros = { 1, 2, 3, 4, 5 };
      ArrayList< Integer > listaEnteros = new ArrayList< Integer >();

      // inserta elementos en listaEnteros
      for ( Integer elemento : enteros )
         listaEnteros.add( elemento );

      System.out.printf( "listaEnteros contiene: %s\n", listaEnteros );
      System.out.printf( "Total de los elementos en listaEnteros: %.0f\n\n",
         sumar( listaEnteros ) );

      // crea, inicializa e imprime en pantalla un objeto ArrayList de objetos Doubles,
      // y después muestra el total de los elementos
      Double[] valoresDouble = { 1.1, 3.3, 5.5 };
      ArrayList< Double > listaDouble = new ArrayList< Double >();

      // inserta elementos en listaDouble
      for ( Double elemento : valoresDouble )
         listaDouble.add( elemento );

      System.out.printf( "listaDouble contiene: %s\n", listaDouble );
      System.out.printf( "Total de los elementos en listaDouble: %.1f\n\n",
         sumar( listaDouble ) );

      // crea, inicializa e imprime en pantalla un objeto ArrayList de objetos Number
      // que contiene objetos Integer y Double, después muestra el total de los elementos
      Number[] numeros = { 1, 2.4, 3, 4.1 }; // objetos Integer y Double
      ArrayList< Number > listaNumeros = new ArrayList< Number >();

      // inserta elementos en listaNumeros
      for ( Number elemento : numeros )
         listaNumeros.add( elemento );

      System.out.printf( "listaNumeros contiene: %s\n", listaNumeros );
      System.out.printf( "Total de los elementos en listaNumeros: %.1f\n",
         sumar( listaNumeros ) );
   } // fin de main

   // calcula el total de los elementos de la pila
   public static double sumar( ArrayList< ? extends Number > lista )
   {
      double total = 0; // inicializa el total

```

Figura 18.15 | Programa de prueba de comodines. (Parte 1 de 2).

```
        // calcula la suma
        for ( Number elemento : lista )
           total += elemento.doubleValue();

        return total;
     } // fin del método sumar
  } // fin de la clase PruebaComodin
```

```
listaEnteros contiene: [1, 2, 3, 4, 5]
Total de los elementos en listaEnteros: 15

listaDouble contiene: [1.1, 3.3, 5.5]
Total de los elementos en listaDouble: 9.9

listaNumeros contiene: [1, 2.4, 3, 4.1]
Total de los elementos en listaNumeros: 10.5
```

Figura 18.15 | Programa de prueba de comodines. (Parte 2 de 2).

18.9 Genéricos y herencia: observaciones

Los genéricos pueden utilizarse con la herencia de varias formas:

- Una clase genérica puede derivarse de una clase no genérica. Por ejemplo, la clase `Object` es una superclase directa o indirecta de toda clase genérica.
- Una clase genérica puede derivarse de otra clase genérica. Por ejemplo, la clase genérica `Stack` (en el paquete `java.util`) es una subclase de la clase genérica `Vector` (en el paquete `java.util`). En el capítulo 19, Colecciones, hablaremos sobre estas clases.
- Una clase no genérica puede derivarse de una clase genérica. Por ejemplo, la clase no genérica `Properties` (en el paquete `java.util`) es una subclase de la clase genérica `Hashtable` (en el paquete `java.util`). También veremos estas clases en el capítulo 19.
- Por último, un método genérico en una subclase puede sobrescribir a un método genérico en una superclase, si ambos métodos tienen las mismas firmas.

18.10 Conclusión

En este capítulo se presentaron los genéricos. Usted aprendió a declarar métodos genéricos y clases genéricas. Aprendió cómo se logra la compatibilidad inversa mediante tipos crudos. También aprendió a utilizar comodines en un método genérico o en una clase genérica. En el siguiente capítulo demostraremos las interfaces, clases y algoritmos del marco de trabajo de colecciones de Java. Como veremos, todas las colecciones que se presentarán utilizan las capacidades genéricas que vimos en este capítulo.

18.11 Recursos en Internet y Web

`www.jcp.org/aboutJava/communityprocess/review/jsr014/`

Página de descarga del Proceso comunitario de Java, para el documento de la especificación de genéricos *Adding Generics to the Java Programming Language: Public Draft Specification, Version 2.0.*

`www.angelikalanger.com/GenericsFAQ/JavaGenericsFAQ.html`

Una colección de preguntas frecuentes acerca de los genéricos en Java.

`java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf`

El tutorial *Generics in the Java Programming Language* por Gilad Bracha (el líder de la especificación para JSR-14 y revisor de este libro) introduce los conceptos de los genéricos, con fragmentos de código de ejemplo.

`today.java.net/pub/a/today/2003/12/02/explorations.html`

`today.java.net/pub/a/today/2004/01/15/wildcards.html`

Los artículos *Explorations: Generics, Erasure, and Bridging* y *Explorations: Wildcards in the Generics Specification*, cada uno por William Grosso, presentan las generalidades acerca de las características de los genéricos, y cómo utilizar los comodines.

Resumen

Sección 18.1 Introducción

- Los métodos genéricos permiten a los programadores especificar, con la declaración de un solo método, un conjunto de métodos relacionados.
- Las clases genéricas permiten a los programadores especificar, con la declaración de una sola clase, un conjunto de tipos relacionados.
- Los métodos genéricos y las clases genéricas se encuentran entre las herramientas más poderosas de Java para la reutilización de software con seguridad en los tipos.

Sección 18.2 Motivación para los métodos genéricos

- Los métodos sobrecargados se utilizan a menudo para realizar operaciones similares en distintos tipos de datos.
- Cuando el compilador encuentra la llamada a un método, trata de localizar la declaración de un método que tenga el mismo nombre del método y los mismos parámetros que coincidan con los tipos de los argumentos en la llamada al método.

Sección 18.3 Métodos genéricos: implementación y traducción en tiempo de compilación

- Si las operaciones realizadas por varios métodos sobrecargados son idénticas para cada tipo de argumento, los métodos sobrecargados se pueden codificar en forma más compacta y conveniente, mediante el uso de un método genérico. Usted puede escribir la declaración de un solo método genérico, el cual se puede llamar con argumentos de distintos tipos de datos. Con base en los tipos de los argumentos que se pasan al método genérico, el compilador maneja la llamada a cada método en forma apropiada.
- Todas las declaraciones de métodos genéricos tienen una sección de parámetros de tipo, delimitada por los signos `<` y `>`, que antecede al tipo de valor de retorno del método.
- Cada sección de parámetros de tipo contiene uno o más parámetros de tipo (también llamados parámetros de tipo formal), separados por comas.
- Un parámetro de tipo es un identificador que especifica el nombre de un tipo genérico. Los parámetros de tipo pueden utilizarse como el tipo de valor de retorno, los tipos de los parámetros y los tipos de las variables locales en la declaración de un método genérico, y actúan como receptáculos para los tipos de los argumentos que se pasan al método genérico, los cuales se conocen como argumentos de tipo actuales. Los parámetros de tipo sólo pueden representar tipos de referencias.
- Los nombres utilizados para los parámetros de tipo en la declaración de un método deben coincidir con los que se declaran en la sección de parámetros de tipo. El nombre de un parámetro de tipo se puede declarar sólo una vez en la sección de parámetros de tipo, pero puede aparecer más de una vez en la lista de parámetros del método. Los nombres de los parámetros de tipo no necesitan ser únicos entre distintos métodos genéricos.
- Cuando el compilador encuentra la llamada a un método, determina los tipos de los argumentos y trata de localizar un método con el mismo nombre y parámetros que coincidan con los tipos de los argumentos. Si no hay un método así, el compilador determina si hay una coincidencia inexacta, pero aplicable.
- El operador relacional `>` no se puede utilizar con tipos de referencias. Sin embargo, es posible comparar dos objetos de la misma clase, si esa clase implementa a la interfaz genérica `Comparable` (paquete `java.lang`).
- Los objetos `Comparable` tienen un método `compareTo` que debe devolver `0` si los objetos son iguales, `-1` si el primer objeto es menor que el segundo, o `1` si el primer objeto es mayor que el segundo.
- Todas las clases de envoltura de tipos para los tipos primitivos implementan a `Comparable`.
- Un beneficio de implementar la interfaz `Comparable` es que los objetos `Comparable` pueden utilizarse con los métodos de ordenamiento y búsqueda de la clase `Collections` (paquete `java.util`).
- Cuando el compilador traduce un método genérico en códigos de byte de Java, elimina la sección de parámetros de tipo y reemplaza los parámetros de tipo con tipos actuales. A este proceso se le conoce como borrado. De manera predeterminada, cada parámetro de tipo se reemplaza con su límite superior. De manera predeterminada, el límite superior es de tipo `Object`, a menos que se especifique otra cosa en la sección de parámetros de tipo.

Sección 18.4 Cuestiones adicionales sobre la traducción en tiempo de compilación: métodos que utilizan un parámetro de tipo como tipo de valor de retorno

- Cuando el compilador realiza el borrado en un método que devuelva una variable de tipo, también inserta operaciones de conversión explícitas en frente de cada llamada a un método, para asegurar que el valor devuelto sea del tipo que espera el método que hizo la llamada.

Sección 18.5 Sobrecarga de métodos genéricos

- Un método genérico puede sobrecargarse. Una clase puede proporcionar dos o más métodos genéricos que especifiquen el mismo nombre del método, pero distintos parámetros para el mismo. Un método genérico también puede sobrecargarse mediante métodos no genéricos que tengan el mismo nombre y el mismo número de parámetros. Cuando el compilador encuentra la llamada a un método, busca la declaración del método que coincida en forma más precisa con el nombre del método y los tipos de los argumentos especificados en la llamada.
- Cuando el compilador encuentra la llamada a un método, realiza un proceso de asociación para determinar cuál método debe llamar. El compilador trata de buscar y utilizar una coincidencia precisa, en la cual los nombres del método y los tipos de los argumentos de la llamada al método coincidan con los de una declaración específica del método. Si esto falla, el compilador determina si hay un método genérico disponible que proporcione una coincidencia precisa del nombre del método y los tipos de los argumentos y, de ser así, utiliza ese método genérico.

Sección 18.6 Clases genéricas

- Las clases genéricas proporcionan los medios para describir una clase en forma independiente del tipo. Así, podemos instanciar objetos específicos del tipo de la clase genérica.
- La declaración de una clase genérica es similar a la declaración de una clase no genérica, excepto que el nombre de la clase va seguido de una sección de parámetros de tipo. Al igual que con los métodos genéricos, la sección de parámetros de tipo de una clase genérica puede tener uno o más parámetros de tipo, separados por comas.
- Cuando se compila una clase genérica, el compilador realiza el borrado en los parámetros de tipo de la clase y los reemplaza con sus límites superiores.
- Los parámetros de tipo no se pueden utilizar en las declaraciones `static` de una clase.
- Al instanciar un objeto de una clase genérica, los tipos especificados entre los signos `<` y `>` después del nombre de la clase se conocen como argumentos de tipo. El compilador los utiliza para reemplazar los parámetros de tipo, de manera que pueda realizar la comprobación de tipos e insertar operaciones de conversión, según sea necesario.

Sección 18.7 Tipos crudos (raw)

- Es posible instanciar una clase genérica sin especificar un argumento de tipo. En este caso, se dice que el nuevo objeto de la clase tiene un tipo crudo, lo cual significa que el compilador utiliza de manera implícita el tipo `Object` (o el límite superior del parámetro de tipo) en la clase genérica para cada argumento de tipo.

Sección 18.8 Comodines en métodos que aceptan parámetros de tipo

- El Marco de trabajo Collections de Java proporciona muchas estructuras de datos genéricas y algoritmos para manipular los elementos de esas estructuras de datos. Tal vez la más simple de las estructuras de datos sea la clase `ArrayList`; una estructura de datos tipo arreglo, que puede cambiar su tamaño en forma dinámica.
- La clase `Number` es la superclase de `Integer` y de `Double`.
- El método `add` de la clase `ArrayList` anexa un elemento al final de la colección.
- El método `toString` de la clase `ArrayList` devuelve una cadena de la forma "[*elementos*]", en la cual *elementos* es una lista separada por comas de las representaciones de cadena de los elementos.
- El método `doubleValue` de la clase `Number` obtiene el valor primitivo subyacente de `Number` como un valor `double`.
- Los argumentos tipo comodín nos permiten especificar parámetros de métodos, valores de retorno, variables, etcétera, que actúen como supertipos de los tipos parametrizados. Un argumento de tipo comodín se denota mediante el signo de interrogación (`?`), el cual representa un "tipo desconocido". Un comodín también puede tener un límite superior.
- Debido a que un comodín (`?`) no es el nombre de un parámetro de tipo, no se puede utilizar como nombre de tipo en el cuerpo de un método.
- Si se especifica un comodín sin un límite superior, entonces sólo pueden invocarse los métodos de tipo `Object` en valores del tipo comodín.
- Los métodos que utilizan comodines como argumentos de tipo no pueden usarse para agregar elementos a una colección a la que hace referencia el parámetro.

Sección 18.9 Genéricos y herencia: observaciones

- Una clase genérica puede derivarse de una clase no genérica. Por ejemplo, `Object` es una superclase directa o indirecta de toda clase genérica.
- Una clase genérica puede derivarse de otra clase genérica.
- Una clase no genérica puede derivarse de una clase genérica.

- Un método genérico en una subclase puede sobrescribir a un método genérico en una superclase, si ambos métodos tienen las mismas firmas.

Terminología

? (argumento de tipo comodín)
add, método de ArrayList
alcance de un parámetro de tipo
argumento de tipo
argumentos de tipo actuales
ArrayList, clase
borrado
clase genérica
clase parametrizada
comodín como argumento de tipo
comodín sin un límite superior
comodín (?)
Comparable<T>, interfaz
compareTo, método de Comparable<T>
Double, clase
doubleValue, método de Number
genéricos
Integer, clase
interfaz genérica
límite superior de un comodín
límite superior de un parámetro de tipo
límite superior predeterminado de un parámetro de tipo
método genérico
Number, clase
parámetro de tipo
parámetro de tipo formal
sección de parámetros de tipo
signos < y >
sobrecargar un método genérico
tipo crudo (raw)
tipo parametrizado
toString, método de ArrayList
variable de tipo

Ejercicios de autoevaluación

18.1 Conteste con *verdadero* o *falso* a cada una de las siguientes proposiciones; en caso de ser *falso*, explique por qué.

a) Un método genérico no puede tener el mismo nombre que un método no genérico.
b) Todas las declaraciones de métodos genéricos tienen una sección de parámetros de tipo, la cual va justo antes del nombre del método.
c) Un método genérico puede sobrecargarse mediante otro método genérico con el mismo nombre, pero con distintos parámetros.
d) Un parámetro de tipo puede declararse sólo una vez en la sección de parámetros de tipo, pero puede aparecer más de una vez en la lista de parámetros del método.
e) Los nombres de los parámetros de tipo entre los distintos métodos genéricos deben ser únicos.
f) El alcance del parámetro de tipo de una clase genérica es toda la clase, excepto sus miembros static.

18.2 Complete los siguientes enunciados:

a) Los __________________ y las __________________ le permiten especificar, con la declaración de un solo método, un conjunto de métodos relacionados, o con la declaración de una sola clase, un conjunto de tipos relacionados, respectivamente.
b) Una sección de parámetros de tipo se delimita mediante __________________.
c) Los __________________ de un método genérico se pueden usar para especificar los tipos de los argumentos del método, para especificar el tipo de valor de retorno y para declarar variables dentro del método.
d) La instrucción "Pila pilaObjetos = new Pila();" indica que pilaObjetos almacena ____________.
e) En la declaración de una clase genérica, el nombre de la clase va seguido por un(a) __________________.
f) La sintaxis __________________ especifica que el límite superior de un comodín es de tipo E.

Respuestas a los ejercicios de autoevaluación

18.1 a) Falso. Los métodos genéricos y los no genéricos pueden tener el mismo nombre. Un método genérico puede sobrecargar a otro método genérico con el mismo nombre, pero con distintos parámetros. Un método genérico también puede sobrecargarse si se proporcionan métodos no genéricos con el mismo nombre del método y el mismo número de argumentos. b) Falso. Todas las declaraciones de métodos tienen una sección de parámetros de tipo que va justo antes

del tipo de valor de retorno del método. c) Verdadero. d) Verdadero. e) Falso. Los nombres de los parámetros de tipo entre los distintos métodos genéricos no tienen que ser únicos. f) Verdadero.

18.2 a) Métodos genéricos, clases genéricas. b) los signos < y >. c) parámetros de tipo. d) un tipo crudo (raw). e) sección de parámetros de tipo. f) ? extends E.

Ejercicios

18.3 Explique el uso de la siguiente notación en un programa en Java:

```
public class Array< T > { }
```

18.4 Escriba un método genérico llamado ordenamientoSeleccion con base en el programa de ordenamiento de las figuras 16.6 y 16.7. Escriba un programa de prueba que introduzca, ordene e imprima en pantalla un arreglo Integer y un arreglo Float. [*Sugerencia:* use <T extends Comparable< T >> en la sección de parámetros de tipo para el método ordenamientoSeleccion, para que pueda utilizar el método compareTo para comparar los objetos de dos tipos genéricos T].

18.5 Sobrecargue el método genérico imprimirArreglo de la figura 18.3 para que reciba dos argumentos enteros adicionales, subindiceInferior y subindiceSuperior. Una llamada a este método debe imprimir sólo la parte designada del arreglo. Valide subindiceInferior y subindiceSuperior. Si cualquiera de los dos está fuera de rango, o si subindiceSuperior es menor o igual que subindiceInferior, el método imprimirArreglo sobrecargado debe lanzar una excepción InvalidSubscriptException; en caso contrario, imprimirArreglo debe devolver el número de elementos impresos. Después modifique main para ejecutar ambas versiones de imprimirArreglo en los arreglos arregloInteger, arregloDouble y arregloCharacter. Pruebe todas las capacidades de ambas versiones de imprimirArreglo.

18.6 Sobrecargue el método genérico imprimirArreglo de la figura 18.3 con una versión no genérica que imprima en forma específica un arreglo de cadenas en un formato tabular impecable, como se muestra en los resultados de ejemplo a continuación:

```
El arreglo arregloCadena contiene:
uno       dos        tres      cuatro
cinco     seis       siete     ocho
```

18.7 Escriba una versión genérica simple del método esIgualA que compare sus dos argumentos con el método equals y devuelva true si son iguales, y false en caso contrario. Use este método genérico en un programa que llame a esIgualA con una variedad de tipos integrados, como Object o Integer. ¿Qué resultado obtiene al tratar de ejecutar este programa?

18.8 Escriba una clase genérica llamada Par, que tenga dos parámetros de tipo: F y S, cada uno de los cuales representa el tipo del primer y segundo elementos del par, respectivamente. Agregue métodos *obtener* y *establecer* para los elementos primero y segundo del par. [*Sugerencia:* el encabezado de la clase debe ser public class Par< F, S >].

18.9 Convierta las clases NodoArbol y Arbol de la figura 17.17 en clases genéricas. Para insertar un objeto en un Arbol, el objeto debe compararse con los objetos en los objetos NodoArbol existentes. Por esta razón, las clases NodoArbol y Arbol deben especificar a Comparable< E > como el límite superior del parámetro de tipo de cada clase. Después de modificar las clases NodoArbol y Arbol, escriba una aplicación de prueba para crear tres objetos Arbol: uno que almacene objetos Integer, uno que almacene objetos Double y uno que almacene objetos String. Inserte 10 valores en cada árbol. Después imprima en pantalla los recorridos preorden, inorden y postorden para cada Arbol.

18.10 Modifique su programa de prueba del ejercicio 18.9 para utilizar un método genérico llamado probarArbol, para probar los tres objetos Arbol. El método debe llamarse tres veces, una para cada objeto Arbol.

18.11 ¿Cómo pueden sobrecargarse los métodos genéricos?

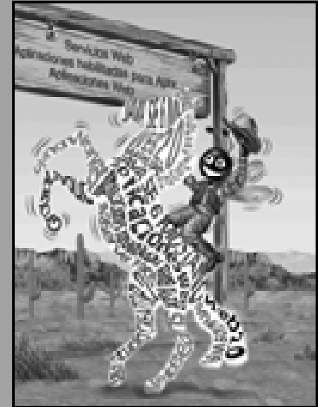
18.12 El compilador realiza un proceso de asociación para determinar cuál método debe llamar al invocar a un método. ¿Bajo qué circunstancias un intento por realizar una asociación produce un error en tiempo de compilación?

18.13 Explique por qué un programa en Java podría utilizar la instrucción

```
ArrayList< Empleado > listaTrabajadores = new ArrayList< Empleado >();
```

19

Colecciones



Creo que ésta es la colección más extraordinaria de talento, de conocimiento humano, que se haya reunido en la Casa Blanca; con la posible excepción de cuando Thomas Jefferson comió solo.

— John F. Kennedy

¡Las figuras que puede alojar un contenedor brillante!

— Theodore Roethke

Un viaje a través de todo el universo en un mapa.

— Miguel de Cervantes

La sabiduría se adquiere, no por la edad, sino por la capacidad.

— Tirus Maccius Plautus

Es un acertijo envuelto en un misterio, dentro de un enigma.

— Winston Churchill

OBJETIVOS

En este capítulo aprenderá a:

- Comprender lo que son las colecciones.
- Utilizar la clase `Arrays` para manipulaciones comunes de arreglos.
- Utilizar las implementaciones del marco de trabajo de colecciones (estructura de datos preempaquetada).
- Utilizar los algoritmos del marco de trabajo de colecciones para manipular varias colecciones (como `search`, `sort` y `fill`).
- Utilizar las interfaces del marco de trabajo de colecciones para programar mediante el polimorfismo.
- Utilizar iteradores para "recorrer" los elementos de una colección.
- Utilizar las tablas hash persistentes que se manipulan con objetos de la clase `Properties`.
- Comprender las envolturas de sincronización y las envolturas modificables.

Plan general

19.1 Introducción

En el capítulo 17 vimos cómo crear y manipular estructuras de datos. La discusión fue de "bajo nivel", en cuanto a que creamos laboriosamente cada elemento de cada estructura de datos en forma dinámica, y modificamos las estructuras de datos manipulando directamente sus elementos y las referencias a ellos. En este capítulo veremos el **marco de trabajo de colecciones** de Java, el cual contiene estructuras de datos, interfaces y algoritmos preempaquetados para manipular esas estructuras de datos. Algunos ejemplos de colecciones son las tarjetas que usted posee en un juego de cartas, su música favorita almacenada en su computadora, los miembros de un equipo deportivo y los registros de bienes raíces en el registro de propiedades de su localidad (que asocia números de libro y página con los propietarios de los bienes inmuebles). En este capítulo también veremos cómo se utilizan los genéricos (vea el capítulo 18) en el marco de trabajo de colecciones de Java.

Con las colecciones, los programadores utilizan las estructuras de datos existentes sin tener que preocuparse por la manera en que éstas se implementan. Éste es un maravilloso ejemplo de reutilización de código. Los programadores pueden codificar más rápido y esperar un excelente rendimiento, maximizando la velocidad de ejecución y minimizando el consumo de memoria. En este capítulo hablaremos sobre las interfaces del marco de trabajo de colecciones que describen las capacidades de cada tipo de colección, las clases de implementación, los algoritmos que procesan a las colecciones y los denominados **iteradores**, junto con la sintaxis de la instrucción `for` mejorada para "recorrer" las colecciones. En este capítulo presentamos una introducción al marco de trabajo de colecciones. Para obtener los detalles completos, visite la página Web `java.sun.com/javase/6/docs/guide/collections`.

El marco de trabajo de colecciones de Java proporciona componentes reutilizables, listos para utilizarse; usted no necesita escribir sus propias clases de colecciones, pero puede hacerlo si lo desea. Estas colecciones están estandarizadas, de manera que las aplicaciones puedan compartirlas fácilmente sin tener que preocuparse por los

detalles relacionados con su implementación. El marco de trabajo de colecciones fomenta aún más la reutilización. A medida que se desarrollen estructuras de datos y algoritmos que se ajusten a este marco de trabajo, una extensa base de programadores estará ya familiarizada con las interfaces y algoritmos implementados por esas estructuras de datos.

19.2 Generalidades acerca de las colecciones

Una **colección** es una estructura de datos (en realidad, un objeto) que puede guardar referencias a otros objetos. Por lo general, las colecciones contienen referencias a objetos, los cuales son todos del mismo tipo. Las interfaces del marco de trabajo de colecciones declaran las operaciones que se deben realizar en forma genérica en varios tipos de colecciones. La figura 19.1 enlista algunas de las interfaces del marco de trabajo de colecciones. Varias implementaciones de estas interfaces se proporcionan dentro del marco de trabajo. Los programadores también pueden proporcionar implementaciones específicas para sus propios requerimientos.

Interfaz	Descripción
`Collection`	La interfaz raíz en la jerarquía de colecciones, a partir de la cual se derivan las interfaces `Set`, `Queue` y `List`.
`Set`	Una colección que no contiene duplicados.
`List`	Una colección ordenada que puede contener elementos duplicados.
`Map`	Asocia claves con valores y no puede contener claves duplicadas.
`Queue`	Por lo general, una colección del tipo primero en entrar, primero en salir, que modela a una línea de espera; pueden especificarse otros órdenes.

Figura 19.1 | Algunas interfaces del marco de trabajo de colecciones.

El marco de trabajo de colecciones proporciona implementaciones de alto rendimiento y alta calidad de las estructuras de datos comunes, y permite la reutilización de software. Estas características minimizan la cantidad de código que necesitan escribir los programadores para crear y manipular colecciones. Las clases y las interfaces del marco de trabajo de colecciones son miembros del paquete `java.util`. En la siguiente sección, comenzaremos nuestra discusión mediante un análisis de las herramientas del marco de trabajo de colecciones para la manipulación de arreglos.

En versiones anteriores de Java, las clases en el marco de trabajo de colecciones almacenaban y manipulaban referencias `Object`. Por ende, podíamos almacenar cualquier objeto en una colección. Un aspecto inconveniente de almacenar referencias `Object` se presenta al obtenerlas de una colección. Por lo general, un programa tiene la necesidad de procesar tipos específicos de objetos. Como resultado, las referencias `Object` que se obtienen de una colección comúnmente necesitan convertirse en un tipo apropiado, para permitir que el programa procese los objetos correctamente.

En Java SE 5, el marco de trabajo de colecciones se mejoró con las herramientas de genéricos que presentamos en el capítulo 18. Esto significa que podemos especificar el tipo exacto que se almacenará en una colección. También recibimos los beneficios de la comprobación de tipos en tiempo de ejecución; el compilador asegura que se utilicen los tipos apropiados con la colección y, si no es así, emite mensajes de error en tiempo de compilación. Además, una vez que especifique el tipo almacenado en una colección, cualquier referencia que obtenga de la colección tendrá el tipo especificado. Esto elimina la necesidad de conversiones de tipo explícitas que pueden lanzar excepciones `ClassCastException`, si el objeto referenciado no es del tipo apropiado. Los programas que se implementaron con versiones anteriores de Java y que utilizan colecciones pueden compilarse de manera apropiada, ya que el compilador utiliza de manera automática los tipos crudos (raw) cuando encuentra colecciones para las cuales no se especificaron argumentos de tipo.

19.3 La clase `Arrays`

La clase **`Arrays`** proporciona métodos `static` para manipular arreglos. En el capítulo 7, nuestra discusión acerca de la manipulación de arreglos fue de nivel bajo, en el sentido en que escribimos el código en sí para ordenar y

buscar en los arreglos. La clase `Arrays` proporciona métodos de alto nivel, como **`sort`** para ordenar un arreglo, `binarySearch` para buscar en un arreglo ordenado, **`equals`** para comparar arreglos y **`fill`** para colocar valores en un arreglo. Estos métodos se sobrecargan para los arreglos de tipo primitivo y los arreglos tipo `Object`. Además, los métodos `sort` y `binarySearch` están sobrecargados con versiones genéricas que permiten a los programadores ordenar y buscar en arreglos que contengan objetos de cualquier tipo. En la figura 19.2 se demuestra el uso de los métodos `fill`, `sort`, `binarySearch` y `equals`. El método `main` (líneas 65 a 85) crea un objeto `UsoArrays` e invoca a sus métodos.

En la línea 17 se hace una llamada al método `static fill` de `Arrays` para llenar los 10 elementos del arreglo `arregloIntLleno` con 7s. Las versiones sobrecargadas de `fill` permiten al programador llenar un rango específico de elementos con el mismo valor.

En la línea 18 se ordenan los elementos del arreglo `arregloDouble`. El método `static sort` de la clase `Arrays` ordena los elementos del arreglo en orden ascendente, de manera predeterminada. Más adelante en este capítulo veremos cómo ordenar elementos en forma descendente. Las versiones sobrecargadas de `sort` permiten al programador ordenar un rango específico de elementos.

En las líneas 21 y 22 se copia el arreglo `arregloInt` en el arreglo `copiaArregloInt`. El primer argumento (`arregloInt`) que se pasa al método **`arraycopy`** de `System` es el arreglo a partir del cual se van a copiar los elementos. El segundo argumento (`0`) es el índice que especifica el punto de inicio en el rango de elementos que se van a copiar del arreglo. Este valor puede ser cualquier índice de arreglo válido. El tercer argumento (`copiaArregloInt`) especifica el arreglo de destino que almacenará la copia. El cuarto argumento (`0`) especifica el índice en el arreglo de destino en donde deberá guardarse el primer elemento copiado. El último argumento especifica el número de elementos a copiar del arreglo en el primer argumento. En este caso copiaremos todos los elementos en el arreglo.

En la línea 50 se hace una llamada al método estático `binarySearch` de la clase `Arrays` para realizar una búsqueda binaria en `arregloInt`, utilizando `valor` como la clave. Si se encuentra `valor`, `binarySearch` devuelve el índice del elemento; en caso contrario, `binarySearch` devuelve un valor negativo. El valor negativo devuelto se basa en el **punto de inserción** de la clave de búsqueda: el índice en donde se insertaría la clave en el arreglo si se fuera a realizar una operación de inserción. Una vez que `binarySearch` determina el punto de inserción, cambia el signo de éste a negativo y le resta `1` para obtener el valor de retorno. Por ejemplo, en la figura 19.2, el punto de inserción para el valor `8763` es el elemento en el arreglo con el índice `6`. El método `binarySearch` cambia el punto de inserción a `–6`, le resta `1` y devuelve el valor `–7`. Al restar 1 al punto de inserción se garantiza que el método `binarySearch` devuelva valores positivos (>= 0) sí, y sólo si se encuentra la clave. Este valor de retorno es útil para agregar elementos en un arreglo ordenado. En el capítulo 16, Búsqueda y ordenamiento, se habla sobre la búsqueda binaria con detalle.

```
1  // Fig. 19.2: UsoArrays.java
2  // Uso de arreglos en Java.
3  import java.util.Arrays;
4
5  public class UsoArrays
6  {
7     private int arregloInt[] = { 1, 2, 3, 4, 5, 6 };
8     private double arregloDouble[] = { 8.4, 9.3, 0.2, 7.9, 3.4 };
9     private int arregloIntLleno[], copiaArregloInt[];
10
11    // el constructor inicializa los arreglos
12    public UsoArrays()
13    {
14       arregloIntLleno = new int[ 10 ]; // crea arreglo int con 10 elementos
15       copiaArregloInt = new int[ arregloInt.length ];
16
17       Arrays.fill( arregloIntLleno, 7 ); // llena con 7s
18       Arrays.sort( arregloDouble ); // ordena arregloDouble en forma ascendente
19
```

Figura 19.2 | Métodos de la clase `Arrays`. (Parte 1 de 3).

```
      // copia el arreglo arregloInt en el arreglo copiaArregloInt
      System.arraycopy( arregloInt, 0, copiaArregloInt,
         0, arregloInt.length );
   } // fin del constructor de UsoArrays

   // imprime los valores en cada arreglo
   public void imprimirArreglos()
   {
      System.out.print( "arregloDouble: " );
      for ( double valorDouble : arregloDouble )
         System.out.printf( "%.1f ", valorDouble );

      System.out.print( "\narregloInt: " );
      for ( int valorInt : arregloInt )
         System.out.printf( "%d ", valorInt );

      System.out.print( "\narregloIntLleno: " );
      for ( int valorInt : arregloIntLleno )
         System.out.printf( "%d ", valorInt );

      System.out.print( "\ncopiaArregloInt: " );
      for ( int valorInt : copiaArregloInt )
         System.out.printf( "%d ", valorInt );

      System.out.println( "\n" );
   } // fin del método imprimirArreglos

   // busca un valor en el arreglo arregloInt
   public int buscarUnInt( int valor )
   {
      return Arrays.binarySearch( arregloInt, valor );
   } // fin del método buscarUnInt

   // compara el contenido del arreglo
   public void imprimirIgualdad()
   {
      boolean b = Arrays.equals( arregloInt, copiaArregloInt );
      System.out.printf( "arregloInt %s copiaArregloInt\n",
         ( b ? "==" : "!=" ) );

      b = Arrays.equals( arregloInt, arregloIntLleno );
      System.out.printf( "arregloInt %s arregloIntLleno\n",
         ( b ? "==" : "!=" ) );
   } // fin del método imprimirIgualdad

   public static void main( String args[] )
   {
      UsoArrays usoArreglos = new UsoArrays();

      usoArreglos.imprimirArreglos();
      usoArreglos.imprimirIgualdad();

      int ubicacion = usoArreglos.buscarUnInt( 5 );
      if ( ubicacion >= 0 )
         System.out.printf(
            "Se encontro el 5 en el elemento %d de arregloInt\n", ubicacion );
      else
         System.out.println( "No se encontro el 5 en arregloInt" );

```

Figura 19.2 | Métodos de la clase `Arrays`. (Parte 2 de 3).

```
79         ubicacion = usoArreglos.buscarUnInt( 8763 );
80         if ( ubicacion >= 0 )
81            System.out.printf(
82               "Se encontro el 8763 en el elemento %d en arregloInt\n", ubicacion );
83         else
84            System.out.println( "No se encontro el 8763 en arregloInt" );
85      } // fin de main
86   } // fin de la clase UsoArrays
```

```
arregloDouble: 0.2 3.4 7.9 8.4 9.3
arregloInt: 1 2 3 4 5 6
arregloIntLleno: 7 7 7 7 7 7 7 7 7 7
copiaArregloInt: 1 2 3 4 5 6

arregloInt == copiaArregloInt
arregloInt != arregloIntLleno
Se encontro el 5 en el elemento 4 de arregloInt
No se encontro el 8763 en arregloInt
```

Figura 19.2 | Métodos de la clase `Arrays`. (Parte 3 de 3).



Error común de programación 19.1

Pasar un arreglo desordenado al método `binarySearch` es un error lógico; el valor devuelto es indefinido.

En las líneas 56 y 60 se hace una llamada al método `static equals` de la clase `Arrays` para determinar si los elementos de dos arreglos son equivalentes. Si los arreglos contienen los mismos elementos en el mismo orden, el método devuelve `true`; en caso contrario, devuelve `false`. La igualdad de cada elemento se compara mediante el uso del método `equals` de `Object`. Muchas clases redefinen el método `equals` para realizar las comparaciones de una manera específica a esas clases. Por ejemplo, la clase `String` declara a `equals` para comparar los caracteres individuales en los dos objetos `String` que se están comparando. Si el método `equals` no se sobrescribe, se utiliza la versión original del método `equals` heredado de la clase `Object`.

19.4 La interfaz `Collection` y la clase `Collections`

La interfaz **`Collection`** es la interfaz raíz en la jerarquía de colecciones, a partir de la cual se derivan las interfaces `Set`, `Queue` y `List`. La interfaz **`Set`** define a una colección que no contiene duplicados. La interfaz **`Queue`** define a una colección que representa a una línea de espera; por lo general, las inserciones se realizan en la parte final de una cola y las eliminaciones en su parte inicial, aunque pueden especificarse otros órdenes. En las secciones 19.8 y 19.9 hablaremos sobre `Queue` y `Set`, respectivamente. La interfaz `Collections` contiene **operaciones masivas** (es decir, operaciones que se llevan a cabo en toda una colección) para agregar, borrar, comparar y retener objetos (o elementos) en una colección. Un objeto `Collection` también puede convertirse en un arreglo. Además, la interfaz `Collection` proporciona un método que devuelve un objeto **`Iterator`**, el cual permite a un programa recorrer toda la colección y eliminar elementos de la misma durante la iteración. En la sección 19.5.1 hablaremos sobre la clase `Iterator`. Otros métodos de la interfaz `Collection` permiten a un programa determinar el tamaño de una colección, y si está vacía o no.



Observación de ingeniería de software 19.1

`Collection` se utiliza comúnmente como un tipo de parámetro de métodos para permitir el procesamiento polimórfico de todos los objetos que implementen a la interfaz `Collection`.



Observación de ingeniería de software 19.2

La mayoría de las implementaciones de colecciones proporcionan un constructor que toma un argumento `Collection`, permitiendo así que se construya una nueva colección, la cual contiene los elementos de la colección especificada.

La clase **Collections** proporciona métodos static que manipulan las colecciones mediante el polimorfismo. Estos métodos implementan algoritmos para buscar, ordenar, etcétera. En el capítulo 16, Búsqueda y ordenamiento, se describieron e implementaron varios algoritmos de búsqueda y ordenamiento. En la sección 19.6 hablaremos más acerca de los algoritmos disponibles en la clase Collections. También cubriremos los **métodos de envoltura** de la clase Collections, los cuales nos permiten tratar a una colección como una **colección sincronizada** (sección 19.2) o una **colección no modificable** (sección 19.13). Las colecciones no modificables son útiles cuando un cliente de una clase necesita ver los elementos de una colección, pero no se le debe permitir que modifique la colección, agregando y eliminando elementos. Las colecciones sincronizadas son para usarse con una poderosa herramienta conocida como subprocesamiento múltiple (que veremos en el capítulo 23). El subprocesamiento múltiple permite a los programas realizar operaciones en paralelo. Cuando dos o más subprocesos de un programa comparten una colección, existe la probabilidad de que ocurran problemas. Como una breve analogía, considere una intersección de tráfico. No podemos permitir que todos los automóviles accedan a una intersección al mismo tiempo; si lo hiciéramos, ocurrirían accidentes. Por esta razón, se proporcionan semáforos en las intersecciones para controlar el acceso a cada intersección. De manera similar, podemos sincronizar el acceso a una colección para asegurar que sólo un subproceso manipule la colección a la vez. Los métodos de envoltura de sincronización de la clase Collections devuelven las versiones sincronizadas de las colecciones que pueden compartirse entre los subprocesos en un programa.

19.5 Listas

Un objeto List (conocido como **secuencia**) es un objeto Collection ordenado que puede contener elementos duplicados. Al igual que los índices de arreglos, los índices de objetos List empiezan desde cero (es decir, el índice del primer elemento es cero). Además de los métodos de interfaz heredados de Collection, List proporciona métodos para manipular elementos a través de sus índices, para manipular un rango especificado de elementos, para buscar elementos y para obtener un objeto **ListIterator** para acceder a los elementos.

La interfaz List es implementada por varias clases, incluyendo a **ArrayList**, **LinkedList** y **Vector**. La conversión autoboxing ocurre cuando se agregan valores de tipo primitivo a objetos de estas clases, ya que sólo almacenan referencias a objetos. Las clases ArrayList y Vector son implementaciones de un objeto List como arreglos que pueden modificar su tamaño. La clase LinkedList es una implementación de la interfaz List como una lista enlazada.

El comportamiento y las herramientas de la clase ArrayList son similares a las de la clase Vector. La principal diferencia entre Vector y ArrayList es que los objetos de la clase Vector están sincronizados de manera predeterminada, mientras que los objetos de la clase ArrayList no. Además, la clase Vector es de Java 1.0, antes de que se agregara el marco de trabajo de colecciones a Java. Como tal, Vector tiene varios métodos que no forman parte de la interfaz List y que no se implementan en la clase ArrayList, pero realizan tareas idénticas. Por ejemplo, los métodos addElement y add de Vector anexan un elemento a un objeto Vector, pero sólo el método add está especificado en la interfaz List y se implementa mediante ArrayList. Las colecciones desincronizadas proporcionan un mejor rendimiento que las sincronizadas. Por esta razón, ArrayList se prefiere comúnmente a Vector en programas que no comparten una colección entre subprocesos.



Tip de rendimiento 19.1

Los objetos ArrayList se comportan igual que los objetos Vector desincronizados y, por lo tanto, se ejecutan con más rapidez que los objetos Vector, ya que los objetos ArrayList no tienen la sobrecarga que implica la sincronización de los subprocesos.



Observación de ingeniería de software 19.3

Los objetos LinkedList pueden usarse para crear pilas, colas, árboles y "colas con dos partes finales" (conocidas en inglés como "deque"). El marco de trabajo de colecciones proporciona implementaciones de algunas de estas estructuras de datos.

En las siguientes tres subsecciones se demuestran las herramientas de List y Collection con varios ejemplos. La sección 19.5.1 se enfoca en eliminar elementos de un objeto ArrayList mediante un objeto Iterator. La sección 19.5.2 se enfoca en ListIterator y varios métodos específicos de List y de LinkedList. La sección 19.5.3 introduce más métodos de List y varios métodos específicos de Vector.

19.5.1 ArrayList e Iterator

En la figura 19.3 se utiliza un objeto `ArrayList` para demostrar varias herramientas de la interfaz `Collection`. El programa coloca dos arreglos `Color` en objetos `ArrayList` y utiliza un objeto `Iterator` para eliminar los elementos en la segunda colección `ArrayList` de la primera colección `ArrayList`.

En las líneas 10 a 13 se declaran e inicializan dos variables arreglo `String`, las cuales se declaran como `final`, por lo que siempre hacen referencia a estos arreglos. Recuerde que es una buena práctica de programación declarar constantes con las palabras clave `static` y `final`. En las líneas 18 y 19 se crean objetos `ArrayList` y se asignan sus referencias a las variables `lista` y `eliminarLista`, respectivamente. Estas dos listas almacenan objetos `String`. Observe que `ArrayList` es una clase genérica a partir de Java SE 5, por lo que podemos especificar un argumento de tipo (`String` en este caso) para indicar el tipo de los elementos en cada lista. Tanto `lista` como `eliminarLista` son colecciones de objetos `String`. En las líneas 22 y 23 se llena `lista` con objetos `String` almacenados en el arreglo `colores`, y en las líneas 26 y 27 se llena `eliminarLista` con objetos `String` almacenados en el arreglo `eliminarColores`, usando el método **`add`** de `List`. En las líneas 32 y 33 se imprime en pantalla cada elemento de `lista`. En la línea 32 se llama al método **`size`** de `List` para obtener el número de elementos del objeto `ArrayList`. En la línea 33 se utiliza el método **`get`** de `List` para obtener valores de elementos individuales. En las líneas 32 y 33 se pudo haber usado la instrucción `for` mejorada. En la línea 36 se hace una llamada al método `eliminarColores` (líneas 46 a 57), y recibe a `lista` y `eliminarLista` como argumentos. El método `eliminarColores` elimina los objetos `String` especificados en `eliminarLista` de la colección `lista`. En las líneas 41 y 42 se imprimen en pantalla los elementos de `lista`, una vez que `eliminarColores` elimina los objetos `String` especificados en `eliminarLista` de la `lista`.

El método `eliminarColores` declara dos parámetros de tipo `Collection` (línea 47), los cuales contienen cadenas que se van a pasar como argumentos a este método. El método accede a los elementos del primer objeto `Collection` (`coleccion1`) mediante un objeto **`Iterator`**. En la línea 50 se llama al método `iterator` de `Collection`, el cual obtiene un objeto `Iterator` para el objeto `Collection`. Observe que las interfaces `Collection` e `Iterator` son tipos genéricos. En la condición del ciclo de continuación (línea 53) se hace una llamada al método **`hasNext`** de `Iterator` para determinar si el objeto `Collection` contiene más elementos. El método `hasNext` devuelve `true` si otro elemento existe, y devuelve `false` en caso contrario.

La condición del `if` en la línea 55 llama al método **`next`** de `Iterator` para obtener una referencia al siguiente elemento, y después utiliza el método **`contains`** del segundo objeto `Collection` (`coleccion2`) para determinar si `coleccion2` contiene el elemento devuelto por `next`. De ser así, en la línea 56 se hace una llamada al método **`remove`** de `Iterator` para eliminar el elemento del objeto `coleccion1` de `Collection`.



Error común de programación 19.2

Si se modifica una colección mediante uno de sus métodos, después de crear un iterador para esa colección, el iterador se vuelve inválido de manera inmediata; cualquier operación realizada con el iterador después de este punto lanza excepciones `ConcurrentModificationException`. Por esta razón, se dice que los iteradores son de "falla rápida".

```
1  // Fig. 19.3: PruebaCollection.java
2  // Uso de la interfaz Collection.
3  import java.util.List;
4  import java.util.ArrayList;
5  import java.util.Collection;
6  import java.util.Iterator;
7
8  public class PruebaCollection
9  {
10    private static final String[] colores =
11       { "MAGENTA", "ROJO", "BLANCO", "AZUL", "CYAN" };
12    private static final String[] eliminarColores =
13       { "ROJO", "BLANCO", "AZUL" };
14
15    // crea objeto ArrayList, le agrega los colores y lo manipula
16    public PruebaCollection()
```

Figura 19.3 | Demostración de la interfaz `Collection` mediante un objeto `ArrayList`. (Parte 1 de 2).

```
    {
       List< String > lista = new ArrayList< String >();
       List< String > eliminarLista = new ArrayList< String >();

       // agrega los elementos en el arreglo colores a la lista
       for ( String color : colores )
          lista.add( color );

       // agrega los elementos en eliminarColores a eliminarLista
       for ( String color : eliminarColores )
          eliminarLista.add( color );

       System.out.println( "ArrayList: " );

       // imprime en pantalla el contenido de la lista
       for ( int cuenta = 0; cuenta < lista.size(); cuenta++ )
          System.out.printf( "%s ", lista.get( cuenta ) );

       // elimina los colores contenidos en eliminarLista
       eliminarColores( lista, eliminarLista );

       System.out.println( "\n\nArrayList despues de llamar a eliminarColores: " );

       // imprime en pantalla el contenido de la lista
       for ( String color : lista )
          System.out.printf( "%s ", color );
    } // fin del constructor de PruebaCollection

    // elimina de coleccion1 los colores especificados en coleccion2
    private void eliminarColores(
       Collection< String > coleccion1, Collection< String > coleccion2 )
    {
       // obtiene el iterador
       Iterator< String > iterador = coleccion1.iterator();

       // itera mientras la colección tenga elementos
       while ( iterador.hasNext() )

          if ( coleccion2.contains( iterador.next() ) )
             iterador.remove(); // elimina el color actual
    } // fin del método eliminarColores

    public static void main( String args[] )
    {
       new PruebaCollection();
    } // fin de main
 } // fin de la clase PruebaCollection
```

```
ArrayList:
MAGENTA ROJO BLANCO AZUL CYAN

ArrayList despues de llamar a eliminarColores:
MAGENTA CYAN
```

Figura 19.3 | Demostración de la interfaz `Collection` mediante un objeto `ArrayList`. (Parte 2 de 2).

19.5.2 LinkedList

En la figura 19.4 se demuestran las operaciones con objetos `LinkedList`. El programa crea dos objetos `LinkedList` que contienen objetos `String`. Los elementos de un objeto `List` se agregan al otro. Después, todos los objetos `String` se convierten a mayúsculas, y se elimina un rango de elementos.

```
// Fig. 19.4: PruebaList.java
// Uso de objetos LinkList.
import java.util.List;
import java.util.LinkedList;
import java.util.ListIterator;

public class PruebaList
{
   private static final String colores[] = { "negro", "amarillo",
      "verde", "azul", "violeta", "plateado" };
   private static final String colores2[] = { "dorado", "blanco",
      "cafe", "azul", "gris", "plateado" };

   // establece y manipula objetos LinkedList
   public PruebaList()
   {
      List< String > lista1 = new LinkedList< String >();
      List< String > lista2 = new LinkedList< String >();

      // agrega elementos a la lista enlace
      for ( String color : colores )
         lista1.add( color );

      // agrega elementos a la lista enlace2
      for ( String color : colores2 )
         lista2.add( color );

      lista1.addAll( lista2 ); // concatena las listas
      lista2 = null; // libera los recursos
      imprimirLista( lista1 ); // imprime los elementos de lista1

      convertirCadenasAMayusculas( lista1 ); // convierte cadena a mayúsculas
      imprimirLista( lista1 ); // imprime los elementos de lista1

      System.out.print( "\nEliminando elementos 4 a 6...");
      eliminarElementos( lista1, 4, 7 ); // elimina los elementos 4 a 7 de la lista
      imprimirLista( lista1 ); // imprime los elementos de lista1
      imprimirListaInversa( lista1 ); // imprime la lista en orden inverso
   } // fin del constructor de PruebaList

   // imprime el contenido del objeto List
   public void imprimirLista( List< String > lista )
   {
      System.out.println( "\nlista: " );

      for ( String color : lista )
         System.out.printf( "%s ", color );

      System.out.println();
   } // fin del método imprimirLista

   // localiza los objetos String y los convierte a mayúsculas
   private void convertirCadenasAMayusculas( List< String > lista )
   {
      ListIterator< String > iterador = lista.listIterator();

      while ( iterador.hasNext() )
      {
         String color = iterador.next();  // obtiene elemento
```

Figura 19.4 | Objetos `List` y `ListIterator`. (Parte 1 de 2).

```
60            iterador.set( color.toUpperCase() ); // convierte a mayúsculas
61         } // fin de while
62      } // fin del método convertirCadenasAMayusculas
63
64      // obtiene sublista y utiliza el método clear para eliminar los elementos de la misma
65      private void eliminarElementos( List< String > lista, int inicio, int fin )
66      {
67         lista.subList( inicio, fin ).clear();  // elimina los elementos
68      } // fin del método eliminarElementos
69
70      // imprime la lista inversa
71      private void imprimirListaInversa( List< String > lista )
72      {
73         ListIterator< String > iterador = lista.listIterator( lista.size() );
74
75         System.out.println( "\nLista inversa:" );
76
77         // imprime la lista en orden inverso
78         while ( iterador.hasPrevious() )
79            System.out.printf( "%s ", iterador.previous() );
80      } // fin del método imprimirListaInversa
81
82      public static void main( String args[] )
83      {
84         new PruebaList();
85      } // fin de main
86   } // fin de la clase PruebaList
```

```
lista:
negro amarillo verde azul violeta plateado dorado blanco cafe azul gris plateado

lista:
NEGRO AMARILLO VERDE AZUL VIOLETA PLATEADO DORADO BLANCO CAFE AZUL GRIS PLATEADO

Eliminando elementos 4 a 6...
lista:
NEGRO AMARILLO VERDE AZUL BLANCO CAFE AZUL GRIS PLATEADO

Lista inversa:
PLATEADO GRIS AZUL CAFE BLANCO AZUL VERDE AMARILLO NEGRO
```

Figura 19.4 | Objetos `List` y `ListIterator`. (Parte 2 de 2).

En las líneas 17 y 18 se crean los objetos `LinkedList` llamados `lista1` y `lista2` de tipo `String`. Observe que `LinkedList` es una clase genérica que tiene un parámetro de tipo, para el cual especificamos el argumento de tipo `String` en este ejemplo. En las líneas 21 a 26 se hace una llamada al método `add` de `List` para anexar elementos de los arreglos `colores` y `colores2` al final de `lista1` y `lista2`, respectivamente.

En la línea 28 se hace una llamada al método **`addAll`** de `List` para anexar todos los elementos de `lista2` al final de `lista1`. En la línea 29 se establece `lista2` en `null`, de manera que el objeto `LinkedList` al que hacía referencia `lista2` pueda marcarse para la recolección de basura. En la línea 30 se hace una llamada al método `imprimirLista` (líneas 42 a 50) para mostrar el contenido de `lista1`. En la línea 32 se hace una llamada al método `convertirCadenaAMayusculas` (líneas 53 a 62) para convertir cada elemento `String` a mayúsculas, y después en la línea 33 se hace una llamada nuevamente a `imprimirLista` para mostrar los objetos `String` modificados. En la línea 36 se hace una llamada al método `eliminarElementos` (líneas 65 a 68) para eliminar los elementos empezando desde el índice 4 hasta, pero no incluyendo, el índice 7 de la lista. En la línea 38 se hace una llamada al método `imprimirListaInversa` (líneas 71 a 80) para imprimir la lista en orden inverso.

El método `convertirCadenasAMayusculas` (líneas 53 a 62) cambia los elementos `String` en minúsculas del argumento `List` por objetos `String` en mayúsculas. En la línea 55 se hace una llamada al método **`list-`**

`Iterator` de `List` para obtener un **iterador bidireccional** (es decir, un iterador que pueda recorrer un objeto `Lista` hacia delante o hacia atrás) para el objeto `List`. Observe que `ListIterator` es una clase genérica. En este ejemplo, el objeto `ListIterator` contiene objetos `String`, ya que el método `listIterator` se llama en un objeto `List` que contiene objetos `String`. En la condición del ciclo `while` (línea 57) se hace una llamada al método `hasNext` para determinar si el objeto `List` contiene otro elemento. En la línea 59 se obtiene el siguiente objeto `String` en el objeto `List`. En la línea 60 se hace una llamada al método **`toUpperCase`** de `String` para obtener una versión en mayúsculas del objeto `String` y se hace una llamada al método **`set`** de `Iterator` para reemplazar el objeto `String` actual al que hace referencia `iterador` con el objeto `String` devuelto por el método **`toUpperCase`**. Al igual que el método **`toUpperCase`**, el método `toLowerCase` de `String` devuelve una versión del objeto `String` en minúsculas.

El método `eliminarElementos` (líneas 65 a 68) elimina un rango de elementos de la lista. En la línea 67 se hace una llamada al método **`subList`** de `List` para obtener una porción del objeto `List` (lo que se conoce como **sublista**). La sublista es simplemente otra vista hacia el interior del objeto `List` desde el que se hace la llamada a `subList`. El método `subList` recibe dos argumentos: el índice inicial para la sublista y el índice final. Observe que el índice final no forma parte del rango de la sublista. En este ejemplo, pasamos `4` (en la línea 36) para el índice inicial y `7` para el índice final a `subList`. La sublista devuelta es el conjunto de elementos con los índices `4` a `6`. A continuación, el programa hace una llamada al método **`clear`** de `List` en la sublista para eliminar los elementos que ésta contiene del objeto `List`. Cualquier cambio realizado a una sublista se hace realmente en el objeto `List` original.

El método `imprimirListaInversa` (líneas 71 a 80) imprime la lista al revés. En la línea 73 se hace una llamada al método **`listIterator`** de `List` con un argumento que especifica la posición inicial (en nuestro caso, el último elemento en la lista) para obtener un iterador bidireccional para la lista. El método **`size`** de `List` devuelve el número de elementos en el objeto `List`. En la condición del ciclo `while` (línea 78) se hace una llamada al método **`hasPrevious`** para determinar si hay más elementos mientras se recorre la lista hacia atrás. En la línea 79 se obtiene el elemento anterior de la lista y se envía como salida al flujo de salida estándar.

Una característica importante del marco de trabajo de colecciones es la habilidad de manipular los elementos de un tipo de colección (como un conjunto) a través de un tipo de colección distinto (como una lista), sin importar la implementación interna de la colección. Al conjunto de métodos `public` a través de los cuales se manipulan las colecciones se le conoce como **vista**.

La clase `Arrays` proporciona el método `static` **`asList`** para ver un arreglo como una colección **`List`** (que encapsula el comportamiento similar al de las listas enlazadas que creamos en el capítulo 17). Una vista `List` permite al programador manipular el arreglo como si fuera una lista. Esto es útil para agregar los elementos en un arreglo a una colección (por ejemplo, un objeto `LinkedList`) y para ordenar los elementos del arreglo. En el siguiente ejemplo le demostraremos cómo crear un objeto `LinkedList` con una vista `List` de un arreglo, ya que no podemos pasar el arreglo a un constructor de `LinkedList`. En la figura 19.9 se demuestra cómo ordenar elementos de un arreglo con una vista `List`. Cualquier modificación realizada a través de la vista `List` cambia el arreglo, y cualquier modificación realizada al arreglo cambia la vista `List`. La única operación permitida en la vista devuelta por `asList` es *establecer*, la cual cambia el valor de la vista y del arreglo de soporte. Cualquier otro intento por cambiar la vista (como agregar o eliminar elementos) produce una excepción **`UnsupportedOperationException`**.

En la figura 19.5 se utiliza el método `asList` para ver un arreglo como una colección `List`, y el método **`toArray`** de un objeto `List` para obtener un arreglo de una colección `LinkedList`. El programa llama al método `asList` para crear una vista `List` de un arreglo, la cual se utiliza después para crear un objeto `LinkedList`, agrega una serie de cadenas a un objeto `LinkedList` y llama al método `toArray` para obtener un arreglo que contiene referencias a esas cadenas. Observe que el instanciamiento de `LinkedList` (línea 13) indica que es una clase genérica que acepta un argumento de tipo: `String`, en este ejemplo.

En las líneas 13 y 14 se construye un objeto `LinkedList` de objetos `String`, el cual contiene los elementos del arreglo `colores`, y se asigna la referencia `LinkedList` a `enlaces`. Observe el uso del método `asList` de `Arrays` para devolver una vista del arreglo como un objeto `List`, y después inicializar el objeto `LinkedList` con el objeto `List`. En la línea 16 se hace una llamada al método **`addLast`** de `LinkedList` para agregar `"rojo"` al final de `enlaces`. En las líneas 17 y 18 se hace una llamada al método **`add`** de `LinkedList` para agregar `"rosa"` como el último elemento y `"verde"` como el elemento en el índice `3` (es decir, el cuarto elemento). Observe que el método **`addLast`** (línea 16) es idéntico en función al método `add` (línea 17). En la línea 19 se hace una llamada al método **`addFirst`** de `LinkedList` para agregar `"cyan"` como el nuevo primer elemento en el objeto

LinkedList. Las operaciones add están permitidas debido a que operan en el objeto LinkedList, no en la vista devuelta por asList. [*Nota:* cuando se agrega "cyan" como el primer elemento, "verde" se convierte en el quinto elemento en el objeto LinkedList].

En la línea 22 se hace una llamada al método toArray de List para obtener un arreglo String de enlaces. El arreglo es una copia de los elementos de la lista; si se modifica el contenido del arreglo no se modifica la lista. El arreglo que se pasa al método toArray debe ser del mismo tipo que se desee que devuelva el método toArray. Si el número de elementos en el arreglo es mayor o igual que el número de elementos en el objeto LinkedList, toArray copia los elementos de la lista en su argumento tipo arreglo y devuelve ese arreglo. Si el objeto LinkedList tiene más elementos que el número de elementos en el arreglo que se pasa a toArray, este método asigna un nuevo arreglo del mismo tipo que recibe como argumento, copia los elementos de la lista en el nuevo arreglo y devuelve este nuevo arreglo.

```
1  // Fig. 19.5: UsoToArray.java
2  // Uso del método toArray.
3  import java.util.LinkedList;
4  import java.util.Arrays;
5
6  public class UsoToArray
7  {
8     // el constructor crea un objeto LinkedList, le agrega elementos y lo convierte en
      arreglo
9     public UsoToArray()
10    {
11       String colores[] = { "negro", "azul", "amarillo" };
12
13       LinkedList< String > enlaces =
14          new LinkedList< String >( Arrays.asList( colores ) );
15
16       enlaces.addLast( "rojo" );  // lo agrega como último elemento
17       enlaces.add( "rosa" );      // lo agrega al final
18       enlaces.add( 3, "verde" );  // lo agrega en el 3er índice
19       enlaces.addFirst( "cyan" ); // lo agrega como primer elemento
20
21       // obtiene los elementos de LinkedList como un arreglo
22       colores = enlaces.toArray( new String[ enlaces.size() ] );
23
24       System.out.println( "colores: " );
25
26       for ( String color : colores )
27          System.out.println( color );
28    } // fin del constructor de UsoToArray
29
30    public static void main( String args[] )
31    {
32       new UsoToArray();
33    } // fin de main
34 } // fin de la clase UsoToArray
```

```
colores:
cyan
negro
azul
amarillo
verde
rojo
rosa
```

Figura 19.5 | Método toArray de List.



Error común de programación 19.3

Pasar un arreglo que contenga datos al método `toArray` *puede crear errores lógicos. Si el número de elementos en el arreglo es menor que el número de elementos en la lista en la que se llama a* `toArray`*, se asigna un nuevo arreglo para almacenar los elementos de la lista (sin preservar los elementos del arreglo). Si el número de elementos en el arreglo es mayor que el número de elementos en la lista, los elementos del arreglo (empezando en el índice cero) se sobrescriben con los elementos de la lista. Los elementos de arreglos que no se sobrescriben retienen sus valores.*

19.5.3 Vector

Al igual que `ArrayList`, la clase `Vector` proporciona las herramientas de las estructuras de datos tipo arreglo que pueden cambiar su tamaño en forma dinámica. Recuerde que el comportamiento y las herramientas de la clase `ArrayList` son similares a las de la clase `Vector`, excepto que los objetos `ArrayList` no proporcionan sincronización de manera predeterminada. Aquí veremos la clase `Vector`, principalmente debido a que es la superclase de la clase `Stack`, la cual se presenta en la sección 19.7.

En cualquier momento, un objeto `Vector` contiene un número de elementos que es menor o igual que su **capacidad**. La capacidad es el espacio que se ha reservado para los elementos del objeto `Vector`. Si un `Vector` requiere capacidad adicional, crece en base a un **incremento de capacidad** que el programador especifica, o mediante un incremento de capacidad predeterminado. Si no especificamos un incremento de capacidad, o si especificamos uno que sea menor o igual a cero, el sistema duplicará el tamaño de un objeto `Vector` cada vez que se necesite capacidad adicional.



Tip de rendimiento 19.2

Insertar un elemento en un objeto `Vector` *cuyo tamaño actual sea menor que su capacidad es una operación relativamente rápida.*



Tip de rendimiento 19.3

Insertar un elemento en un objeto `Vector` *que necesita crecer más para dar cabida al nuevo elemento es una operación relativamente lenta.*



Tip de rendimiento 19.4

El incremento de capacidad predeterminado duplica el tamaño del objeto `Vector`*. Esto puede parecer un desperdicio de almacenamiento, pero en realidad es una manera eficiente para que muchos objetos* `Vector` *aumenten rápidamente al "tamaño correcto". Esta operación es mucho más eficiente que aumentar el objeto* `Vector` *cada vez sólo el espacio necesario para contener un solo elemento. La desventaja es que el objeto* `Vector` *podría ocupar más espacio de lo requerido. Éste es un clásico ejemplo de la concesión entre espacio y tiempo.*



Tip de rendimiento 19.5

Si el almacenamiento es primordial, use el método `trimToSize` *de* `Vector` *para recortar la capacidad del objeto* `Vector` *a su tamaño exacto. Esta operación optimiza el uso que da un objeto* `Vector` *al almacenamiento. Sin embargo, al agregar otro elemento al objeto* `Vector`*, éste se verá forzado a crecer en forma dinámica (de nuevo, una operación relativamente lenta); al recortar su tamaño mediante* `trimToSize`*, no queda espacio para que crezca.*

En la figura 19.6 se demuestra la clase `Vector` y varios de sus métodos. Para obtener información completa acerca de la clase `Vector`, visite el sitio Web `java.sun.com/javase/6/docs/api/java/util/Vector.html`.

El constructor de la aplicación crea un objeto `Vector` (línea 12) de tipo `String`, con una capacidad inicial de 10 elementos y un incremento de capacidad de cero (los valores predeterminados para un objeto `Vector`). Observe que `Vector` es una clase genérica, la cual recibe un argumento que especifica el tipo de los elementos almacenados en el objeto `Vector`. Un incremento de capacidad de cero indica que este objeto `Vector` duplicará su tamaño cada vez que necesite crecer, para dar cabida a más elementos. La clase `Vector` proporciona otros tres constructores. El constructor que recibe un argumento entero crea un objeto `Vector` vacío con la **capacidad inicial** especificada por ese argumento. El constructor que recibe dos argumentos crea un objeto `Vector` con la capacidad inicial especificada por el primer argumento, y el **incremento de capacidad** especificado por el segundo argumento. Cada vez que el vector necesita crecer, agrega espacio para el número especificado de elementos en

```java
// Fig. 19.6: PruebaVector.java
// Uso de la clase Vector.
import java.util.Vector;
import java.util.NoSuchElementException;

public class PruebaVector
{
   private static final String colores[] = { "rojo", "blanco", "azul" };

   public PruebaVector()
   {
      Vector< String > vector = new Vector< String >();
      imprimirVector( vector ); // imprime el vector

      // agrega elementos al vector
      for ( String color : colores )
         vector.add( color );

      imprimirVector( vector ); // imprime el vector

      // imprime los elementos primero y último
      try
      {
         System.out.printf( "Primer elemento: %s\n", vector.firstElement());
         System.out.printf( "Ultimo elemento: %s\n", vector.lastElement() );
      } // fin de try
      // atrapa la excepción si el vector está vacío
      catch ( NoSuchElementException excepcion )
      {
         excepcion.printStackTrace();
      } // fin de catch

      // ¿el vector contiene "rojo"?
      if ( vector.contains( "rojo" ) )
         System.out.printf( "\se encontro \"rojo\" en el indice %d\n\n",
            vector.indexOf( "rojo" ) );
      else
         System.out.println( "\no se encontro \"rojo\"\n" );

      vector.remove( "rojo" ); // elimina la cadena "rojo"
      System.out.println( "se elimino \"rojo\" );
      imprimirVector( vector ); // imprime el vector

      // ¿el vector contiene "rojo" después de la operación de eliminación?
      if ( vector.contains( "rojo" ) )
         System.out.printf(
            “se encontro \"rojo\" en el indice %d\n", vector.indexOf( "rojo" ) );
      else
         System.out.println( "no se encontro \"rojo\"" );

      // imprime el tamaño y la capacidad del vector
      System.out.printf( "\nTamanio: %d\nCapacidad: %d\n", vector.size(),
         vector.capacity() );
   } // fin del constructor de PruebaVector

   private void imprimirVector( Vector< String > vectorAImprimir )
   {
      if ( vectorAImprimir.isEmpty() )
         System.out.print( "el vector esta vacio" ); // vectorAImprimir está vacío
```

Figura 19.6 | La clase `Vector` del paquete `java.util`. (Parte 1 de 2).

```
60         else  // itera a través de los elementos
61         {
62            System.out.print( "el vector contiene: " );
63
64            // imprime los elementos
65            for ( String elemento : vectorAImprimir )
66               System.out.printf( "%s ", elemento );
67         } // fin de else
68
69         System.out.println( "\n" );
70      } // fin del método imprimirVector
71
72      public static void main( String args[] )
73      {
74         new PruebaVector(); // crea objeto y llama a su constructor
75      } // fin de main
76   } // fin de la clase PruebaVector
```

```
el vector esta vacio

el vector contiene: rojo blanco azul

Primer elemento: rojo
Ultimo elemento: azul

se encontro "rojo" en el indice 0

se elimino "rojo"
el vector contiene: blanco azul

no se encontro "rojo"

Tamanio: 2
Capacidad: 10
```

Figura 19.6 | La clase `Vector` del paquete `java.util`. (Parte 2 de 2).

el incremento de capacidad. El constructor que recibe un objeto `Collection` crea una copia de los elementos de una colección y los almacena en el objeto `Vector`.

En la línea 17 se hace una llamada al método **`add`** de `Vector` para agregar objetos (en este programa son de tipo `String`) al final del objeto `Vector`. Si es necesario, el objeto `Vector` incrementa su capacidad para dar cabida al nuevo elemento. La clase `Vector` también proporciona un método **`add`** que recibe dos argumentos. Este método recibe un objeto y un entero, e inserta el objeto en el índice especificado en el objeto `Vector`. El método **`set`** reemplaza el elemento en una posición especificada en el objeto `Vector`, con un elemento especificado. El método **`insertElementAt`** proporciona la misma funcionalidad que el método `add` que recibe dos argumentos, excepto que el orden de los parámetros se invierte.

En la línea 24 se hace una llamada al método **`firstElement`** de `Vector` para devolver una referencia al primer elemento en el objeto `Vector`. En la línea 25 se hace una llamada al método **`lastElement`** de `Vector` para devolver una referencia al último elemento en el objeto `Vector`. Cada uno de estos métodos lanza una excepción `NoSuchElementException` si no hay elementos en el objeto `Vector` cuando se hace la llamada al método.

En la línea 34 se hace una llamada al método **`contains`** de `Vector` para determinar si el objeto `Vector` contiene `"rojo"`. El método devuelve `true` si su argumento está en el objeto `Vector`; en caso contrario, el método devuelve `false`. El método `contains` utiliza el método `equals` de `Object` para determinar si la clave de búsqueda es igual a uno de los elementos del objeto `Vector`. Muchas clases sobrescriben el método `equals` para realizar las comparaciones de una manera específica para esas clases. Por ejemplo, la clase `String` declara a `equals` para comparar los caracteres individuales en los dos objetos `String` que se van a comparar. Si el método `equals` no se sobrescribe, se utiliza la versión original del método `equals` heredada de la clase `Object`.



Error común de programación 19.4

Sin sobrescribir el método `equals`, el programa realiza comparaciones mediante el operador == para determinar si dos referencias se refieren al mismo objeto en la memoria.

En la línea 36 se hace una llamada al método **`indexOf`** para determinar el índice de la primera ubicación en el objeto `Vector` que contenga el argumento. El método devuelve `-1` si el argumento no se encuentra en el objeto `Vector`. Una versión sobrecargada de este método recibe un segundo argumento que especifica el índice en el objeto `Vector` en el que debe empezar la búsqueda.



Tip de rendimiento 19.6

Los métodos de `Vector` llamados `contains` e `indexOf` realizan búsquedas lineales en el contenido de un objeto `Vector`. Estas búsquedas son ineficientes para objetos `Vector` más grandes. Si un programa busca frecuentemente elementos en una colección, considere utilizar una de las implementaciones de `Map` de la API Collection de Java (sección 19.10), la cual proporciona herramientas de búsqueda de alta velocidad.

En la línea 40 se hace una llamada al método **`remove`** de `Vector` para eliminar del objeto `Vector` la primera ocurrencia de su argumento. Este método devuelve `true` si encuentra el elemento en el objeto `Vector`; en caso contrario, el método devuelve `false`. Si se elimina el elemento, todos los elementos después de ese elemento en el objeto `Vector` se desplazan una posición hacia el inicio del objeto `Vector`, para llenar la posición del elemento eliminado. La clase `Vector` también proporciona el método **`removeAllElements`** para eliminar todos los elementos de un objeto `Vector`, y el método **`removeElementAt`** para eliminar el elemento en un índice especificado.

En las líneas 52 y 53 se utilizan los métodos **`size`** y **`capacity`** de `Vector` para determinar el número de elementos actuales en el `Vector`, y el número de elementos que pueden almacenarse en el `Vector` sin asignar más memoria, respectivamente.

En la línea 58 se hace una llamada al método **`isEmpty`** de `Vector` para determinar si el objeto `Vector` está vacío. El método devuelve `true` si no hay elementos en el objeto `Vector`; en caso contrario, el método devuelve `false`. En las líneas 65 y 66 se utiliza la instrucción `for` mejorada para imprimir todos los elementos en el vector.

Entre los métodos introducidos en la figura 19.6, `firstElement`, `lastElement` y `capacity` sólo se pueden utilizar con `Vector`. Los otros métodos (por ejemplo, `add`, `contains`, `indexOf`, `remove`, `size` e `isEmpty`) se declaran mediante `List`, lo cual significa que cualquier clase que implemente a `List` (como `Vector`) puede utilizarlos.

19.6 Algoritmos de las colecciones

El marco de trabajo de colecciones cuenta con varios algoritmos de alto rendimiento para manipular los elementos de una colección. Estos algoritmos se implementan como métodos `static` de la clase `Collections` (figura 19.7). Los algoritmos `sort`, `binarySearch`, `reverse`, `shuffle`, `fill` y `copy` operan con objetos `List`. Los algoritmos `min`, `max`, `addAll`, `frequency` y `disjoint` operan con objetos `Collections`.

Algoritmo	Descripción
`sort`	Ordena los elementos de un objeto `List`.
`binarySearch`	Localiza un objeto en un objeto `List`.
`reverse`	Invierte los elementos de un objeto `List`.
`shuffle`	Ordena al azar los elementos de un objeto `List`.
`fill`	Establece cada elemento de un objeto `List` para que haga referencia a un objeto especificado.
`copy`	Copia referencias de un objeto `List` a otro.
`min`	Devuelve el elemento más pequeño en un objeto `Collection`.

Figura 19.7 | Algoritmos de colecciones. (Parte 1 de 2).

Algoritmo	Descripción
`max`	Devuelve el elemento más grande en un objeto `Collection`.
`addAll`	Anexa todos los elementos en un arreglo a una colección.
`frequency`	Calcula cuántos elementos en la colección son iguales al elemento especificado.
`disjoint`	Determina si dos colecciones no tienen elementos en común.

Figura 19.7 | Algoritmos de colecciones. (Parte 2 de 2).



Observación de ingeniería de software 19.4

Los algoritmos del marco de trabajo de colecciones son polimórficos. Es decir, cada algoritmo puede operar en objetos que implementen interfaces específicas, sin importar sus implementaciones subyacentes.

19.6.1 El algoritmo `sort`

El algoritmo **`sort`** ordena los elementos de un objeto `List`, el cual debe implementar a la interfaz **`Comparable`**. El orden se determina en base al orden natural del tipo de los elementos, según su implementación mediante el método `compareTo` de ese objeto. El método `compareTo` está declarado en la interfaz `Comparable` y algunas veces se le conoce como el **método de comparación natural**. La llamada a `sort` puede especificar como segundo argumento un objeto **`Comparator`**, para determinar un ordenamiento alterno de los elementos.

Ordenamiento ascendente

En la figura 19.8 se utiliza el algoritmo `sort` para ordenar los elementos de un objeto `List` en forma ascendente (línea 20). Recuerde que `List` es un tipo genérico y acepta un argumento de tipo, el cual especifica el tipo de elemento de lista; en la línea 15 se declara a `lista` como un objeto `List` de objetos `String`. Observe que en las líneas 18 y 23 se utiliza una llamada implícita al método `toString` de `lista` para imprimir el contenido de la lista en el formato que se muestra en las líneas segunda y cuarta de los resultados.

Ordenamiento descendente

En la figura 19.9 se ordena la misma lista de cadenas utilizadas en la figura 19.8, en orden descendente. El ejemplo introduce la interfaz `Comparator`, la cual se utiliza para ordenar los elementos de un objeto `Collection` en un orden distinto. En la línea 21 se hace una llamada al método `sort` de `Collections` para ordenar el objeto `List` en orden descendente. El método `static` **`reverseOrder`** de `Collections` devuelve un objeto `Comparator` que ordena los elementos de la colección en orden inverso.

```
1  // Fig. 19.8: Ordenamiento1.java
2  // Uso del algoritmo sort.
3  import java.util.List;
4  import java.util.Arrays;
5  import java.util.Collections;
6
7  public class Ordenamiento1
8  {
9     private static final String palos[] =
10       { "Corazones", "Diamantes", "Bastos", "Espadas" };
11
12    // muestra los elementos del arreglo
13    public void imprimirElementos()
14    {
15       List< String > lista = Arrays.asList( palos ); // crea objeto List
```

Figura 19.8 | El método `sort` de `Collections`. (Parte 1 de 2).

```

      // imprime lista
      System.out.printf( "Elementos del arreglo desordenados:\n%s\n", lista );

      Collections.sort( lista ); // ordena ArrayList

      // imprime lista
      System.out.printf( "Elementos del arreglo ordenados:\n%s\n", lista );
   } // fin del método imprimirElementos

   public static void main( String args[] )
   {
      Ordenamiento1 orden1 = new Ordenamiento1();
      orden1.imprimirElementos();
   } // fin de main
} // fin de la clase Ordenamiento1
```

```
Elementos del arreglo desordenados:
[Corazones, Diamantes, Bastos, Espadas]
Elementos del arreglo ordenados:
[Bastos, Corazones, Diamantes, Espadas]
```

Figura 19.8 | El método `sort` de `Collections`. (Parte 2 de 2).

```
// Fig. 19.9: Ordenamiento2.java
// Uso de un objeto Comparator con el algoritmo sort.
import java.util.List;
import java.util.Arrays;
import java.util.Collections;

public class Ordenamiento2
{
   private static final String palos[] =
      { "Corazones", "Diamantes", "Bastos", "Espadas" };

   // imprime los elementos del objeto List
   public void imprimirElementos()
   {
      List< String > lista = Arrays.asList( palos ); // crea objeto List

      // imprime los elementos del objeto List
      System.out.printf( "Elementos del arreglo desordenados:\n%s\n", lista );

      // ordena en forma descendente, utilizando un comparador
      Collections.sort( lista, Collections.reverseOrder() );

      // imprime los elementos del objeto List
      System.out.printf( "Elementos de lista ordenados:\n%s\n", lista );
   } // fin del método imprimirElementos

   public static void main( String args[] )
   {
      Ordenamiento2 ordenamiento = new Ordenamiento2();
      ordenamiento.imprimirElementos();
    } // fin de main
} // fin de la clase Ordenamiento2
```

Figura 19.9 | El método `sort` de `Collections` con un objeto `Comparator`. (Parte 1 de 2).

```
Elementos del arreglo desordenados:
[Corazones, Diamantes, Bastos, Espadas]
Elementos de lista ordenados:
[Espadas, Diamantes, Corazones, Bastos]
```

Figura 19.9 | El método `sort` de `Collections` con un objeto `Comparator`. (Parte 2 de 2).

Ordenamiento mediante un objeto `Comparator`

En la figura 19.10 se crea una clase `Comparator` personalizada, llamada `ComparadorTiempo`, la cual implementa a la interfaz `Comparator` para comparar dos objetos `Tiempo2`. La clase `Tiempo2`, declarada en la figura 8.5, representa tiempos con horas, minutos y segundos.

La clase `ComparadorTiempo` implementa a la interfaz `Comparator`, un tipo genérico que recibe un argumento (en este caso, `Tiempo2`). El método `compare` (líneas 7 a 26) realiza comparaciones entre objetos `Tiempo2`. En la línea 9 se comparan las dos horas de los objetos `Tiempo2`. Si las horas son distintas (línea 12), entonces devolvemos este valor. Si el valor es positivo, entonces la primera hora es mayor que la segunda y el primer tiempo es mayor que el segundo. Si este valor es negativo, entonces la primera hora es menor que la segunda y el primer tiempo es menor que el segundo. Si este valor es cero, las horas son iguales y debemos evaluar los minutos (y tal vez los segundos) para determinar cuál tiempo es mayor.

En la figura 19.11 se ordena una lista mediante el uso de la clase `Comparator` personalizada, llamada `ComparadorTiempo`. En la línea 11 se crea un objeto `ArrayList` de objetos `Tiempo2`. Recuerde que `ArrayList` y `List` son tipos genéricos y aceptan un argumento de tipo que especifica el tipo de los elementos de la colección. En las líneas 13 a 17 se crean cinco objetos `Tiempo2` y se agregan a esta lista. En la línea 23 se hace una llamada al método `sort`, y le pasamos un objeto de nuestra clase `ComparadorTiempo` (figura 19.10).

```
1  // Fig. 19.10: ComparadorTiempo.java
2  // Clase Comparator personalizada que compara dos objetos Tiempo2.
3  import java.util.Comparator;
4
5  public class ComparadorTiempo implements Comparator< Tiempo2 >
6  {
7     public int compare( Tiempo2 tiempo1, Tiempo2 tiempo2 )
8     {
9        int compararHora = tiempo1.obtenerHora() - tiempo2.obtenerHora(); // compara la hora
10
11       // evalúa la hora primero
12       if ( compararHora != 0 )
13          return compararHora;
14
15       int comparaMinuto =
16          tiempo1.obtenerMinuto() - tiempo2.obtenerMinuto(); // compara el minuto
17
18       // después evalúa el minuto
19       if ( comparaMinuto != 0 )
20          return comparaMinuto;
21
22       int compararSegundo =
23          tiempo1.obtenerSegundo() - tiempo2.obtenerSegundo(); // compara el segundo
24
25       return compararSegundo; // devuelve el resultado de comparar los segundos
26    } // fin del método compare
27 } // fin de la clase ComparadorTiempo
```

Figura 19.10 | Clase `Comparator` personalizada que compara dos objetos `Tiempo2`.

```java
// Fig. 19.11: Ordenamiento3.java
// Ordena una lista usando la clase Comparator personalizada ComparadorTiempo.
import java.util.List;
import java.util.ArrayList;
import java.util.Collections;

public class Ordenamiento3
{
   public void imprimirElementos()
   {
      List< Tiempo2 > lista = new ArrayList< Tiempo2 >(); // crea objeto List

      lista.add( new Tiempo2(  6, 24, 34 ) );
      lista.add( new Tiempo2( 18, 14, 58 ) );
      lista.add( new Tiempo2(  6, 05, 34 ) );
      lista.add( new Tiempo2( 12, 14, 58 ) );
      lista.add( new Tiempo2(  6, 24, 22 ) );

      // imprime los elementos del objeto List
      System.out.printf( "Elementos del arreglo desordenados:\n%s\n", lista );

      // ordena usando un comparador
      Collections.sort( lista, new ComparadorTiempo() );

      // imprime los elementos del objeto List
      System.out.printf( "Elementos de la lista ordenados:\n%s\n", lista );
   } // fin del método imprimirElementos

   public static void main( String args[] )
   {
      Ordenamiento3 ordenamiento3 = new Ordenamiento3();
      ordenamiento3.imprimirElementos();
   } // fin de main
} // fin de la clase Ordenamiento3
```

```
Elementos del arreglo desordenados:
[6:24:34 AM, 6:14:58 PM, 6:05:34 AM, 12:14:58 PM, 6:24:22 AM]
Elementos de la lista ordenados:
[6:05:34 AM, 6:24:22 AM, 6:24:34 AM, 12:14:58 PM, 6:14:58 PM]
```

Figura 19.11 | El método `sort` de `Collections` con un objeto `Comparator` personalizado.

19.6.2 El algoritmo shuffle

El algoritmo **shuffle** ordena al azar los elementos de un objeto `List`. En el capítulo 7 presentamos una simulación para barajar y repartir cartas, en la que se utiliza un ciclo para barajar un mazo de cartas. En la figura 19.12, utilizamos el algoritmo `shuffle` para barajar un mazo de objetos `Carta` que podría usarse en un simulador de juego de cartas.

La clase `Carta` (líneas 8 a 41) representa a una carta en un mazo de cartas. Cada `Carta` tiene una cara y un palo. Las líneas 10 a 12 declaran dos tipos `enum` (`Cara` y `Palo`) que representan la cara y el palo de la carta, respectivamente. El método `toString` (líneas 37 a 40) devuelve un objeto `String` que contiene la cara y el palo de la `Carta`, separados por la cadena `" de "`. Cuando una constante `enum` se convierte en una cadena, el identificador de la constante se utiliza como la representación de cadena. Por lo general, utilizamos letras mayúsculas para las constantes `enum`. En este ejemplo, optamos por usar letras mayúsculas sólo para la primera letra de cada constante `enum`, porque queremos que la carta se muestre con letras iniciales mayúsculas para la cara y el palo (por ejemplo, `"As de Bastos"`).

En las líneas 55 a 62 se llena el arreglo `mazo` con cartas que tienen combinaciones únicas de cara y palo. Tanto `Cara` como `Palo` son tipos `public static enum` de la clase `Carta`. Para usar estos tipos `enum` fuera de la

clase `Carta`, debe calificar el nombre de cada tipo `enum` con el nombre de la clase en la que reside (es decir, `Carta`) y un separador punto (`.`). Así, en las líneas 55 y 57 se utilizan `Carta.Palo` y `Carta.Cara` para declarar las variables de control de las instrucciones `for`. Recuerde que el método `values` de un tipo `enum` devuelve un arreglo que contiene todas las constantes del tipo `enum`. En las líneas 55 a 62 se utilizan instrucciones `for` mejoradas para construir 52 nuevos objetos `Carta`.

```
1  // Fig. 19.12: MazoDeCartas.java
2  // Uso del algoritmo shuffle.
3  import java.util.List;
4  import java.util.Arrays;
5  import java.util.Collections;
6
7  // clase para representar un objeto Carta en un mazo de cartas
8  class Carta
9  {
10    public static enum Cara { As, Dos, Tres, Cuatro, Cinco, Seis,
11       Siete, Ocho, Nueve, Diez, Joto, Quina, Rey  };
12    public static enum Palo { Bastos, Diamantes, Corazones, Espadas };
13
14    private final Cara cara; // cara de la carta
15    private final Palo palo; // palo de la carta
16
17    // constructor con dos argumentos
18    public Carta( Cara caraCarta, Palo paloCarta )
19    {
20       cara = caraCarta; // inicializa la cara de la carta
21       palo = paloCarta; // inicializa el palo de la carta
22    } // fin del constructor de Carta con dos argumentos
23
24    // devuelve la cara de la carta
25    public Cara obtenerCara()
26    {
27       return cara;
28    } // fin del método obtenerCara
29
30    // devuelve el palo de la Carta
31    public Palo obtenerPalo()
32    {
33       return palo;
34    } // fin del método obtenerPalo
35
36    // devuelve la representación String de la Carta
37    public String toString()
38    {
39       return String.format( "%s de %s", cara, palo );
40    } // fin del método toString
41 } // fin de la clase Carta
42
43 // declaración de la clase MazoDeCartas
44 public class MazoDeCartas
45 {
46    private List< Carta > lista; // declara objeto List que almacenará los objetos Carta
47
48    // establece mazo de objetos Carta y baraja
49    public MazoDeCartas()
50    {
51       Carta[] mazo = new Carta[ 52 ];
```

Figura 19.12 | Barajar y repartir cartas con el método `shuffle` de `Collections`. (Parte 1 de 2).

```
      int cuenta = 0; // número de cartas

      // llena el mazo con objetos Carta
      for ( Carta.Palo palo : Carta.Palo.values() )
      {
         for ( Carta.Cara cara : Carta.Cara.values() )
         {
            mazo[ cuenta ] = new Carta( cara, palo );
            cuenta++;
         } // fin de for
      } // fin de for

      lista = Arrays.asList( mazo ); // obtiene objeto List
      Collections.shuffle( lista );   // baraja el mazo
   } // fin del constructor de MazoDeCartas

   // imprime el mazo
   public void imprimirCartas()
   {
      // muestra las 52 cartas en dos columnas
      for ( int i = 0; i < lista.size(); i++ )
         System.out.printf( "%-20s%s", lista.get( i ),
            ( ( i + 1 ) % 2 == 0 ) ? "\n" : "\t" );
   } // fin del método imprimirCartas

   public static void main( String args[] )
   {
      MazoDeCartas cartas = new MazoDeCartas();
      cartas.imprimirCartas();
   } // fin de main
} // fin de la clase MazoDeCartas
```

```
Ocho de Bastos         Siete de Corazones
As de Corazones        Nueve de Espadas
Quina de Espadas       Ocho de Corazones
Cuatro de Corazones    Tres de Diamantes
Dos de Espadas         Seis de Espadas
Nueve de Bastos        Nueve de Corazones
Joto de Bastos         Dos de Diamantes
Nueve de Diamantes     Rey de Corazones
Cinco de Corazones     Ocho de Diamantes
Dos de Bastos          Diez de Diamantes
Diez de Bastos         Seis de Corazones
Cinco de Bastos        Tres de Bastos
Diez de Espadas        Tres de Espadas
Seis de Bastos         Tres de Corazones
Siete de Espadas       As de Espadas
Rey de Espadas         Joto de Corazones
As de Diamantes        Seis de Diamantes
Joto de Diamantes      Cinco de Diamantes
Quina de Diamantes     Cuatro de Espadas
Dos de Corazones       Rey de Bastos
Cinco de Espadas       Cuatro de Bastos
Siete de Diamantes     Cuatro de Diamantes
Quina de Bastos        Diez de Corazones
Joto de Espadas        Quina de Corazones
As de Bastos           Siete de Bastos
Ocho de Espadas        Rey de Diamantes
```

Figura 19.12 | Barajar y repartir cartas con el método `shuffle` de `Collections`. (Parte 2 de 2).

La acción de barajar las cartas ocurre en la línea 65, en la cual se hace una llamada al método `static shuffle` de la clase `Collections` para barajar los elementos del arreglo. El método `shuffle` requiere un argumento `List`, por lo que debemos obtener una vista `List` del arreglo antes de poder barajarlo. En la línea 64 se invoca el método `static asList` de la clase `Arrays` para obtener una vista `List` del arreglo `mazo`.

El método `imprimirCartas` (líneas 69 a 75) muestra el mazo de cartas en dos columnas. En cada iteración del ciclo, en las líneas 73 y 74 se imprime una carta justificada a la izquierda, en un campo de 20 caracteres seguido de una nueva línea o de una cadena vacía, con base en el número de cartas mostradas hasta ese momento. Si el número de cartas es par, se imprime una nueva línea; en caso contrario, se imprime un tabulador.

19.6.3 Los algoritmos reverse, fill, copy, max y min

La clase `Collections` proporciona algoritmos para invertir, llenar y copiar objetos `List`. El algoritmo **`reverse`** invierte el orden de los elementos en un objeto `List` y el algoritmo **`fill`** sobrescribe los elementos en un objeto `List` con un valor especificado. La operación `fill` es útil para reinicializar un objeto `List`. El algoritmo **`copy`** recibe dos argumentos: un objeto `List` de destino y un objeto `List` de origen. Cada elemento del objeto `List` de origen se copia al objeto `List` de destino. El objeto `List` de destino debe tener cuando menos la misma longitud que el objeto `List` de origen; de lo contrario, se producirá una excepción `IndexOutOfBoundsException`. Si el objeto `List` de destino es más largo, los elementos que no se sobrescriban permanecerán sin cambio.

Cada uno de los algoritmos que hemos visto hasta ahora opera en objetos `List`. Los algoritmos **`min`** y **`max`** operan en cualquier objeto `Collection`. El algoritmo `min` devuelve el elemento más pequeño en un objeto `Collection` y el algoritmo `max` devuelve el elemento más grande en un objeto `Collection`. Ambos algoritmos pueden llamarse con un objeto `Comparator` como segundo argumento, para realizar comparaciones personalizadas entre objetos, como el objeto `ComparadorTiempo` en la figura 19.11. En la figura 19.13 se demuestra el uso de los algoritmos `reverse`, `fill`, `copy`, `min` y `max`. Observe que se declara el tipo genérico `List` para almacenar objetos `Character`.

En la línea 24 se hace una llamada al método `reverse` de `Collections` para invertir el orden de `lista`. El método `reverse` recibe un argumento `List`. En este caso, `lista` es una vista `List` del arreglo `letras`. Ahora el arreglo `letras` tiene sus elementos en orden inverso. En la línea 28 se copian los elementos de `lista` en `copiaLista`, usando el método `copy` de `Collections`. Los cambios a `copiaLista` no cambian a `letras`, ya que `copiaLista` es un objeto `List` separado que no es una vista `List` para `letras`. El método `copy` requiere dos argumentos `List`. En la línea 32 se hace una llamada al método `fill` de `Collections` para colocar la cadena `"R"` en cada elemento de `lista`. Como `lista` es una vista `List` de `letras`, esta operación cambia cada elemento en `letras` a `"R"`. El método `fill` requiere un objeto `List` como primer argumento, y un objeto `Object` como segundo argumento. En las líneas 45 y 46 se hace una llamada a los métodos `max` y `min` de `Collections` para buscar el elemento más grande y más pequeño de la colección, respectivamente. Recuerde que un objeto `List` es un objeto `Collection`, por lo que en las líneas 45 y 46 se puede pasar un objeto `List` a los métodos `max` y `min`.

```
1   // Fig. 19.13: Algoritmos1.java
2   // Uso de los algoritmos reverse, fill, copy, min y max.
3   import java.util.List;
4   import java.util.Arrays;
5   import java.util.Collections;
6
7   public class Algoritmos1
8   {
9      private Character[] letras = { 'P', 'C', 'M' };
10     private Character[] copiaLetras;
11     private List< Character > lista;
12     private List< Character > copiaLista;
13
14     // crea un objeto List y lo manipula con los métodos de Collections
15     public Algoritmos1()
16     {
17        lista = Arrays.asList( letras ); // obtiene el objeto List
```

Figura 19.13 | Los métodos `reverse`, `fill`, `copy`, `max` y `min` de `Collections`. (Parte 1 de 2).

```
      copiaLetras = new Character[ 3 ];
      copiaLista = Arrays.asList( copiaLetras ); // vista List de copiaLetras

      System.out.println( "Lista inicial: " );
      imprimir( lista );

      Collections.reverse( lista ); // invierte el orden
      System.out.println( "\nDespues de llamar a reverse: " );
      imprimir( lista );

      Collections.copy( copiaLista, lista ); // copia el objeto List
      System.out.println( "\nDespues de copy: " );
      imprimir( copiaLista );

      Collections.fill( lista, 'R' ); // llena la lista con Rs
      System.out.println( "\nDespues de llamar a fill: " );
      imprimir( lista );
   } // fin del constructor de Algoritmos1

   // imprime la información del objeto List
   private void imprimir( List< Character > refLista )
   {
      System.out.print( "La lista es: " );

      for ( Character elemento : refLista )
         System.out.printf( "%s ", elemento );

      System.out.printf( "\nMax: %s", Collections.max( refLista ) );
      System.out.printf( "  Min: %s\n", Collections.min( refLista ) );
   } // fin del método imprimir

   public static void main( String args[] )
   {
      new Algoritmos1();
   } // fin de main
} // fin de la clase Algoritmos1
```

```
Lista inicial:
La lista es: P C M
Max: P  Min: C

Despues de llamar a reverse:
La lista es: M C P
Max: P  Min: C

Despues de copy:
La lista es: M C P
Max: P  Min: C

Despues de llamar a fill:
La lista es: R R R
Max: R  Min: R
```

Figura 19.13 | Los métodos `reverse`, `fill`, `copy`, `max` y `min` de `Collections`. (Parte 2 de 2).

19.6.4 El algoritmo binarySearch

En la sección 16.2.2 estudiamos el algoritmo de búsqueda binaria, de alta velocidad. Este algoritmo se incluye en el marco de trabajo de colecciones de Java como un método `static` de la clase `Collections`. El algoritmo **`binarySearch`** localiza un objeto en un objeto `List` (es decir, un objeto `LinkedList`, `Vector` o `ArrayList`). Si

se encuentra el objeto, se devuelve el índice de ese objeto. Si no se encuentra el objeto, `binarySearch` devuelve un valor negativo. El algoritmo `binarySearch` determina este valor negativo calculando primero el punto de inserción y cambiando el signo del punto de inserción a negativo. Después, `binarySearch` resta uno al punto de inserción para obtener el valor de retorno, el cual garantiza que el método `binarySearch` devolverá números positivos (>= 0), sí y sólo si se encuentra el objeto. Si varios elementos en la lista coinciden con la clave de búsqueda, no hay garantía de que uno se localice primero. En la figura 19.14 se utiliza el algoritmo `binarySearch` para buscar una serie de cadenas en un objeto `ArrayList`.

```java
// Fig. 19.14: PruebaBusquedaBinaria.java
// Uso del algoritmo binarySearch.
import java.util.List;
import java.util.Arrays;
import java.util.Collections;
import java.util.ArrayList;

public class PruebaBusquedaBinaria
{
   private static final String colores[] = { "rojo", "blanco",
      "azul", "negro", "amarillo", "morado", "carne", "rosa" };
   private List< String > lista; // referencia ArrayList

   // crea, ordena e imprime la lista
   public PruebaBusquedaBinaria()
   {
      lista = new ArrayList< String >( Arrays.asList( colores ) );
      Collections.sort( lista ); // ordena el objeto ArrayList
      System.out.printf( "ArrayList ordenado: %s\n", lista );
   } // fin del constructor de PruebaBusquedaBinaria

   // busca varios valores en la lista
   private void buscar()
   {
      imprimirResultadosBusqueda( colores[ 3 ] ); // primer elemento
      imprimirResultadosBusqueda( colores[ 0 ] ); // elemento medio
      imprimirResultadosBusqueda( colores[ 7 ] ); // último elemento
      imprimirResultadosBusqueda( "aqua" ); // debajo del menor
      imprimirResultadosBusqueda( "gris" ); // no existe
      imprimirResultadosBusqueda( "verdeazulado" ); // no existe
   } // fin del método buscar

   // método ayudante para realizar búsquedas
   private void imprimirResultadosBusqueda( String clave )
   {
      int resultado = 0;

      System.out.printf( "\nBuscando: %s\n", clave );
      resultado = Collections.binarySearch( lista, clave );

      if ( resultado >= 0 )
         System.out.printf( "Se encontro en el indice %d\n", resultado );
      else
         System.out.printf( "No se encontro (%d)\n",resultado );
   } // fin del método imprimirResultadosBusqueda

   public static void main( String args[] )
   {
      PruebaBusquedaBinaria pruebaBusquedaBinaria = new PruebaBusquedaBinaria();
```

Figura 19.14 | El método `binarySearch` de `Collections`. (Parte 1 de 2).

```
50         pruebaBusquedaBinaria.buscar();
51      } // fin de main
52   } // fin de la clase PruebaBusquedaBinaria
```

```
ArrayList ordenado: [amarillo, azul, blanco, carne, morado, negro, rojo, rosa]

Buscando: negro
Se encontro en el indice 5

Buscando: rojo
Se encontro en el indice 6

Buscando: rosa
Se encontro en el indice 7

Buscando: aqua
No se encontro (-2)

Buscando: gris
No se encontro (-5)

Buscando: verdeazulado
No se encontro (-9)
```

Figura 19.14 | El método `binarySearch` de `Collections`. (Parte 2 de 2).

Recuerde que tanto `List` como `ArrayList` son tipos genéricos (líneas 12 y 17). El método `binarySearch` de `Collections` espera que los elementos de la lista estén en orden ascendente, por lo que la línea 18 en el constructor ordena la lista con el método `sort` de `Collections`. Si los elementos de la lista no están ordenados, el resultado es indefinido. En la línea 19 se imprime la lista ordenada en la pantalla. El método `buscar` (líneas 23 a 31) se llama desde `main` para realizar las búsquedas. Cada búsqueda llama al método `imprimirResultadosBusqueda` (líneas 34 a 45) para realizar la búsqueda e imprimir los resultados en pantalla. En la línea 39 se hace una llamada al método `binarySearch` de `Collections` para buscar en `lista` la `clave` especificada. El método `binarySearch` recibe un objeto `List` como primer argumento, y un objeto `Object` como segundo argumento. En las líneas 41 a 44 se imprimen en pantalla los resultados de la búsqueda. Una versión sobrecargada de `binarySearch` recibe un objeto `Comparator` como tercer argumento, el cual especifica la forma en que `binarySearch` debe comparar los elementos.

19.6.5 Los algoritmos addAll, frequency y disjoint

La clase `Collections` también proporciona los algoritmos **`addAll`**, **`frequency`** y **`disjoint`**. El algoritmo `addAll` recibe dos argumentos: un objeto `Collection` en el que se va(n) a insertar el (los) nuevo(s) elemento(s) y un arreglo que proporciona los elementos a insertar. El algoritmo `frequency` recibe dos argumentos: un objeto `Collection` en el que se va a buscar y un objeto `Object` que se va a buscar en la colección. El método `frequency` devuelve el número de veces que aparece el segundo argumento en la colección. El algoritmo `disjoint` recibe dos objetos `Collections` y devuelve `true` si no tienen elementos en común. En la figura 19.15 se demuestra el uso de los algoritmos `addAll`, `frequency` y `disjoint`.

En la línea 19 se inicializa `lista` con los elementos en el arreglo `colores`, y en las líneas 20 a 22 se agregan los objetos `String` `"negro"`, `"rojo"` y `"verde"` a `vector`. En la línea 31 se invoca el método `addAll` para agregar los elementos en el arreglo `colores` a `vector`. En la línea 40 se obtiene la frecuencia del objeto `String` `"rojo"` en el objeto `Collection` llamado `vector`, usando el método `frequency`. Observe que en las líneas 41 y 42 se utiliza el nuevo método `printf` para imprimir la frecuencia en pantalla. En la línea 45 se invoca el método `disjoint` para evaluar si los objetos `Collections` `lista` y `vector` tienen elementos en común.

```java
// Fig. 19.15: Algoritmos2.java
// Uso de los algoritmos addAll, frequency y disjoint.
import java.util.List;
import java.util.Vector;
import java.util.Arrays;
import java.util.Collections;

public class Algoritmos2
{
   private String[] colores = { "rojo", "blanco", "amarillo", "azul" };
   private List< String > lista;
   private Vector< String > vector = new Vector< String >();

   // crea objetos List y Vector
   // y los manipula con métodos de Collections
   public Algoritmos2()
   {
      // inicializa lista y vector
      lista = Arrays.asList( colores );
      vector.add( "negro" );
      vector.add( "rojo" );
      vector.add( "verde" );

      System.out.println( "Antes de addAll, el vector contiene: " );

      // muestra los elementos en el vector
      for ( String s : vector )
         System.out.printf( "%s ", s );

      // agrega los elementos en colores a lista
      Collections.addAll( vector, colores );

      System.out.println( "\n\nDespues de addAll, el vector contiene: " );

      // muestra los elementos en el vector
      for ( String s : vector )
         System.out.printf( "%s ", s );

      // obtiene la frecuencia de "rojo"
      int frecuencia = Collections.frequency( vector, "rojo" );
      System.out.printf(
         "\n\nFrecuencia de rojo en el vector: %d\n", frecuencia );

      // comprueba si lista y vector tienen elementos en común
      boolean desunion = Collections.disjoint( lista, vector );

      System.out.printf( "\nlista y vector %s elementos en comun\n",
         ( desunion ? "no tienen" : "tienen"  ) );
   } // fin del constructor de Algoritmos2

   public static void main( String args[] )
   {
      new Algoritmos2();
   } // fin de main
} // fin de la clase Algoritmos2
```

```
Antes de addAll, el vector contiene:
negro rojo verde
```

Figura 19.15 | Los métodos `addAll`, `frequency` y `disjoint` de `Collections`. (Parte 1 de 2).

```
Despues de addAll, el vector contiene:
negro rojo verde rojo blanco amarillo azul

Frecuencia de rojo en el vector: 2

lista y vector tienen elementos en comun
```

Figura 19.15 | Los métodos `addAll`, `frequency` y `disjoint` de `Collections`. (Parte 2 de 2).

19.7 La clase Stack del paquete java.util

En el capítulo 17, Estructuras de datos, aprendimos a construir estructuras de datos fundamentales, incluyendo listas enlazadas, pilas, colas y árboles. En un mundo de reutilización de software, en vez de construir las estructuras de datos a medida que las necesitamos, podemos a menudo aprovechar las estructuras de datos existentes. En esta sección, investigaremos la clase **`Stack`** en el paquete de utilerías de Java (`java.util`).

En la sección 19.5.3 hablamos sobre la clase `Vector`, la cual implementa a un arreglo que puede cambiar su tamaño en forma dinámica. La clase `Stack` extiende a la clase `Vector` para implementar una estructura de datos tipo pila. La conversión autoboxing ocurre cuando agregamos un tipo primitivo a un objeto `Stack`, ya que la clase `Stack` sólo almacena referencias a objetos. En la figura 19.16 se demuestran varios métodos de `Stack`. Para obtener los detalles de la clase `Stack`, visite el sitio Web `java.sun.com/javase/6/docs/api/java/util/Stack.html`.

```
// Fig. 19.16: PruebaStack.java
// Programa para probar la clase java.util.Stack.
import java.util.Stack;
import java.util.EmptyStackException;

public class PruebaStack
{
   public PruebaStack()
   {
      Stack< Number > pila = new Stack< Number >();

      // crea números para almacenarlos en la pila
      Long numeroLong = 12L;
      Integer numeroInt = 34567;
      Float numeroFloat = 1.0F;
      Double numeroDouble = 1234.5678;

      // usa el método push
      pila.push( numeroLong ); // mete un long
      imprimirPila( pila );
      pila.push( numeroInt ); // mete un int
      imprimirPila( pila );
      pila.push( numeroFloat ); // mete un float
      imprimirPila( pila );
      pila.push( numeroDouble ); // mete un double
      imprimirPila( pila );

      // elimina los elementos de la pila
      try
      {
         Number objetoEliminado = null;

         // saca elementos de la pila
         while ( true )
```

Figura 19.16 | La clase `Stack` del paquete `java.util`. (Parte 1 de 2).

```java
         {
            objetoEliminado = pila.pop(); // usa el método pop
            System.out.printf( "%s se saco\n", objetoEliminado );
            imprimirPila( pila );
         } // fin de while
      } // fin de try
      catch ( EmptyStackException emptyStackException )
      {
         emptyStackException.printStackTrace();
      } // fin de catch
   } // fin del constructor de PruebaStack

   private void imprimirPila( Stack< Number > pila )
   {
      if ( pila.isEmpty() )
         System.out.print( "la pila esta vacia\n\n" ); // la pila está vacía
      else  // la pila no está vacía
      {
         System.out.print( "la pila contiene: " );

         // itera a través de los elementos
         for ( Number numero : pila )
            System.out.printf( "%s ", numero );

         System.out.print( "(superior) \n\n" ); // indica la parte superior de la pila
      } // fin de else
   } // fin del método imprimirPila

   public static void main( String args[] )
   {
      new PruebaStack();
   } // fin de main
} // fin de la clase PruebaStack
```

```
la pila contiene: 12 (superior)

la pila contiene: 12 34567 (superior)

la pila contiene: 12 34567 1.0 (superior)

la pila contiene: 12 34567 1.0 1234.5678 (superior)

1234.5678 se saco
la pila contiene: 12 34567 1.0 (superior)

1.0 se saco
la pila contiene: 12 34567 (superior)

34567 se saco
la pila contiene: 12 (superior)

12 se saco
la pila esta vacia

java.util.EmptyStackException
        at java.util.Stack.peek(Stack.java:85)
        at java.util.Stack.pop(Stack.java:67)
        at PruebaStack.<init>(PruebaStack.java:36)
        at PruebaStack.main(PruebaStack.java:65)
```

Figura 19.16 | La clase Stack del paquete java.util. (Parte 2 de 2).

En la línea 10 del constructor se crea un objeto `Stack` vacío de tipo `Number`. La clase `Number` (en el paquete `java.lang`) es la superclase de la mayoría de las clases de envoltura (como `Integer`, `Double`) para los tipos primitivos. Al crear un objeto `Stack` de objetos `Number`, se pueden meter en la pila objetos de cualquier clase que extienda a la clase `Number`. En cada una de las líneas 19, 21, 23 y 25 se hace una llamada al método **`push`** de `Stack` para agregar objetos a la parte superior de la pila. Observe las literales `12L` (línea 13) y `1.0F` (línea 15). Cualquier literal entera que tenga el **sufijo `L`** es un valor `long`. Cualquier literal entera sin un sufijo es un valor `int`. De manera similar, cualquier literal de punto flotante que tenga el **sufijo `F`** es un valor `float`. Una literal de punto flotante sin un sufijo es un valor `double`. Puede aprender más acerca de las literales numéricas en la *Especificación del lenguaje Java*, en el sitio Web `java.sun.com/docs/books/jls/second_edition/html/expressions.doc.html#224125`.

Un ciclo infinito (líneas 34 a 39) llama al método **`pop`** de `Stack` para eliminar el elemento superior de la pila. El método devuelve una referencia `Number` al elemento eliminado. Si no hay elementos en el objeto `Stack`, el método `pop` lanza una excepción **`EmptyStackException`**, la cual termina el ciclo. La clase `Stack` también declara el método **`peek`**. Este método devuelve el elemento superior de la pila sin sacarlo.

En la línea 49 se hace una llamada al método **`isEmpty`** de `Stack` (heredado por `Stack` de la clase `Vector`) para determinar si la pila está vacía. Si está vacía, el método devuelve `true`; en caso contrario, devuelve `false`.

El método `imprimirPila` (líneas 47 a 61) utiliza la instrucción `for` mejorada para iterar a través de los elementos en la pila. La parte superior actual de la pila (el último valor que se metió a la pila) es el primer valor que se imprime. Como la clase `Stack` extiende a la clase `Vector`, toda la interfaz `public` de la clase `Vector` está disponible para los clientes de la clase `Stack`.



Tip para prevenir errores 19.1

Como `Stack` extiende a `Vector`, todos los métodos `public` de `Vector` pueden llamarse en objetos `Stack`, aún si los métodos no representan operaciones de pila convencionales. Por ejemplo, el método `add` de `Vector` se puede utilizar para insertar un elemento en cualquier parte de una pila; una operación que podría "corromper" los datos de la pila. Al manipular un objeto `Stack`, sólo deben usarse los métodos `push` y `pop` para agregar y eliminar elementos de la pila, respectivamente.

19.8 La clase `PriorityQueue` y la interfaz `Queue`

En la sección 17.8 presentamos la estructura de datos tipo cola y creamos nuestra propia implementación de ella. En esta sección investigaremos la interfaz **`Queue`** y la clase **`PriorityQueue`** del paquete de utilerías de Java (`java.util`). `Queue`, una nueva interfaz de colecciones introducida en Java SE 5, extiende a la interfaz `Collection` y proporciona operaciones adicionales para insertar, eliminar e inspeccionar elementos en una cola. `PriorityQueue`, una de las clases que implementa a la interfaz `Queue`, ordena los elementos en base a su orden natural, según lo especificado por el método `compareTo` de los elementos `Comparable`, o mediante un objeto `Comparator` que se suministra a través del constructor.

La clase `PriorityQueue` proporciona una funcionalidad que permite inserciones en orden en la estructura de datos subyacente, y eliminaciones de la parte frontal de la estructura de datos subyacente. Al agregar elementos a un objeto `PriorityQueue`, los elementos se insertan en orden de prioridad, de tal forma que el elemento con mayor prioridad (es decir, el valor más grande) será el primer elemento eliminado del objeto `PriorityQueue`.

Las operaciones comunes de `PriorityQueue` son: **`offer`** para insertar un elemento en la ubicación apropiada, con base en el orden de prioridad, **`poll`** para eliminar el elemento de mayor prioridad de la cola de prioridad (es decir, la parte inicial o cabeza de la cola), **`peek`** para obtener una referencia al elemento de mayor prioridad de la cola de prioridad (sin eliminar ese elemento), **`clear`** para eliminar todos los elementos en la cola de prioridad y **`size`** para obtener el número de elementos en la cola de prioridad. En la figura 19.17 se demuestra la clase `PriorityQueue`.

En la línea 10 se crea un objeto `PriorityQueue` que almacena objetos `Double` con una capacidad inicial de `11` elementos, y se ordenan los elementos de acuerdo con el ordenamiento natural del objeto (los valores predeterminados para un objeto `PriorityQueue`). Observe que `PriorityQueue` es una clase genérica, y que en la línea 10 se crea una instancia de un objeto `PriorityQueue` con un argumento de tipo `Double`. La clase `PriorityQueue` proporciona cinco constructores adicionales. Uno de éstos recibe un `int` y un objeto `Comparator` para crear un objeto `PriorityQueue` con la capacidad inicial especificada por el valor `int` y el ordenamiento por el objeto `Comparator`. En las líneas 13 a 15 se utiliza el método `offer` para agregar elementos a la cola de prioridad.

El método `offer` lanza una excepción `NullPointException` si el programa trata de agregar un objeto `null` a la cola. El ciclo en las líneas 20 a 24 utiliza el método `size` para determinar si la cola de prioridad está vacía (línea 20). Mientras haya más elementos, en la línea 22 se utiliza el método `peek` de `PriorityQueue` para obtener el elemento de mayor prioridad en la cola, para imprimirlo en pantalla (sin eliminarlo de la cola). En la línea 23 se elimina el elemento de mayor prioridad en la cola, con el método `poll`.

```
1  // Fig. 19.17: PruebaPriorityQueue.java
2  // Programa de prueba de la clase PriorityQueue de la biblioteca estándar.
3  import java.util.PriorityQueue;
4
5  public class PruebaPriorityQueue
6  {
7     public static void main( String args[] )
8     {
9        // cola con capacidad de 11
10       PriorityQueue< Double > cola = new PriorityQueue< Double >();
11
12       // inserta elementos en la cola
13       cola.offer( 3.2 );
14       cola.offer( 9.8 );
15       cola.offer( 5.4 );
16
17       System.out.print( "Sondeando de cola: " );
18
19       // muestra los elementos en la cola
20       while ( cola.size() > 0 )
21       {
22          System.out.printf( "%.1f ", cola.peek() ); // ve el elemento superior
23          cola.poll(); // elimina el elemento superior
24       } // fin de while
25    } // fin de main
26 } // fin de la clase PruebaPriorityQueue
```

```
Sondeando de cola: 3.2 5.4 9.8
```

Figura 19.17 | Programa de prueba de la clase `PriorityQueue`.

19.9 Conjuntos

Un objeto **`Set`** es un objeto `Collection` que contiene elementos únicos (es decir, sin elementos duplicados). El marco de trabajo de colecciones contiene varias implementaciones de `Set`, incluyendo a **`HashSet`** y **`TreeSet`**. `HashSet` almacena sus elementos en una tabla de hash, y `TreeSet` almacena sus elementos en un árbol. El concepto de las tablas de hash se presenta en la sección 19.10. En la figura 19.18 se utiliza un objeto `HashSet` para eliminar las cadenas duplicadas de un objeto `List`. Recuerde que tanto `List` como `Collection` son tipos genéricos, por lo que en la línea 18 se crea un objeto `List` que contiene objetos `String`, y en la línea 24 se pasa un objeto `Collection` de objetos `String` al método `imprimirSinDuplicados`.

El método `imprimirSinDuplicados` (líneas 24 a 35), el cual es llamado desde el constructor, recibe un argumento `Collection`. En la línea 27 se crea un objeto `HashSet` a partir del argumento `Collection`. Observe que tanto `Set` como `HashSet` son tipos genéricos. Por definición, los objetos `Set` no contienen valores duplicados, por lo que cuando se construye el objeto `HashSet`, éste elimina cualquier valor duplicado en el objeto `Collection`. En las líneas 31 y 32 se imprimen en pantalla los elementos en el objeto `Set`.

Conjuntos ordenados

El marco de trabajo de colecciones también incluye la interfaz **`SortedSet`** (que extiende a `Set`) para los conjuntos que mantengan a sus elementos ordenados; ya sea en el orden natural de los elementos (por ejemplo, los

```java
// Fig. 19.18: PruebaSet.java
// Uso de un objeto HashSet para eliminar duplicados.
import java.util.List;
import java.util.Arrays;
import java.util.HashSet;
import java.util.Set;
import java.util.Collection;

public class PruebaSet
{
   private static final String colores[] = { "rojo", "blanco", "azul",
      "verde", "gris", "naranja", "carne", "blanco", "cyan",
      "durazno", "gris", "naranja" };

   // crea e imprime un objeto ArrayList
   public PruebaSet()
   {
      List< String > lista = Arrays.asList( colores );
      System.out.printf( "ArrayList: %s\n", lista );
      imprimirSinDuplicados( lista );
   } // fin del constructor de PruebaSet

   // crea conjunto a partir del arreglo para eliminar duplicados
   private void imprimirSinDuplicados( Collection< String > coleccion )
   {
      // crea un objeto HashSet
      Set< String > conjunto = new HashSet< String >( coleccion );

      System.out.println( "\nLos valores sin duplicados son: " );

      for ( String s : conjunto )
         System.out.printf( "%s ", s );

      System.out.println();
   } // fin del método imprimirSinDuplicados

   public static void main( String args[] )
   {
      new PruebaSet();
   } // fin de main
} // fin de la clase PruebaSet
```

```
ArrayList: [rojo, blanco, azul, verde, gris, naranja, carne, blanco, cyan, durazno, gris,
naranja]

Los valores sin duplicados son:
durazno gris verde azul blanco rojo cyan carne naranja
```

Figura 19.18 | Objeto `HashSet` utilizado para eliminar valores duplicados de un arreglo de cadenas.

números se encuentran en orden ascendente) o en un orden especificado por un objeto `Comparator`. La clase `TreeSet` implementa a `SortedSet`. El programa de la figura 19.19 coloca cadenas en un objeto `TreeSet`. Estas cadenas se ordenan al ser agregadas al objeto `TreeSet`. Este ejemplo también demuestra los **métodos de vista de rango**, los cuales permiten a un programa ver una porción de una colección.

En las líneas 16 y 17 del constructor se crea un objeto `TreeSet` de objetos `String` que contiene los elementos del arreglo `nombres`, y se asigna el objeto `SortedSet` a la variable `arbol`. Tanto `SortedSet` como `TreeSet` son tipos genéricos. En la línea 20 se imprime en pantalla el conjunto inicial de cadenas, utilizando el método `imprimirConjunto` (líneas 36 a 42), sobre el cual hablaremos en breve. En la línea 24 se hace una llamada al método **headSet** de `TreeSet` para obtener un subconjunto del objeto `TreeSet`, en el que todos los elementos

serán menores que `"naranja"`. La vista devuelta de `headSet` se imprime a continuación con `imprimirConjunto`. Si se hace algún cambio al subconjunto, éste se reflejará también en el objeto `TreeSet` original, debido a que el subconjunto devuelto por `headSet` es una vista del objeto `TreeSet`.

En la línea 28 se hace una llamada al método **`tailSet`** de `TreeSet` para obtener un subconjunto en el que cada elemento sea mayor o igual que `"naranja"`, y después se imprime el resultado en pantalla. Cualquier cambio realizado a través de la vista `tailSet` se realiza también en el objeto `TreeSet` original. En las líneas 31 y 32 se hace una llamada a los métodos **`first`** y **`last`** de `SortedSet` para obtener el elemento más pequeño y más grande del conjunto, respectivamente.

```
1  // Fig. 19.19: PruebaSortedSet.java
2  // Uso de TreeSet y SortedSet.
3  import java.util.Arrays;
4  import java.util.SortedSet;
5  import java.util.TreeSet;
6
7  public class PruebaSortedSet
8  {
9     private static final String nombres[] = { "amarillo", "verde",
10        "negro", "carne", "gris", "blanco", "naranja", "rojo", "verde" };
11
12    // crea un conjunto ordenado con TreeSet, y después lo manipula
13    public PruebaSortedSet()
14    {
15       // crea objeto TreeSet
16       SortedSet< String > arbol =
17          new TreeSet< String >( Arrays.asList( nombres ) );
18
19       System.out.println( "conjunto ordenado:  " );
20       imprimirConjunto( arbol ); // imprime el contenido del arbol
21
22       // obtiene subconjunto mediante headSet, con base en "naranja"
23       System.out.print( "\nheadSet (\"naranja\"):  " );
24       imprimirConjunto( arbol.headSet( "naranja" ) );
25
26       // obtiene subconjunto mediante tailSet, con base en "naranja"
27       System.out.print( "tailSet (\"naranja\"):  " );
28       imprimirConjunto( arbol.tailSet( "naranja" ) );
29
30       // obtiene los elementos primero y último
31       System.out.printf( "primero: %s\n", arbol.first() );
32       System.out.printf( "ultimo : %s\n", arbol.last() );
33    } // fin del constructor de PruebaSortedSet
34
35    // imprime el conjunto en pantalla
36    private void imprimirConjunto( SortedSet< String > conjunto )
37    {
38       for ( String s : conjunto )
39          System.out.print( "%s " , s );
40
41       System.out.println();
42    } // fin del método imprimirConjunto
43
44    public static void main( String args[] )
45    {
46       new PruebaSortedSet();
47    } // fin de main
48 } // fin de la clase PruebaSortedSet
```

Figura 19.19 | Uso de objetos `SortedSet` y `TreeSet`. (Parte 1 de 2).

```
conjunto ordenado:
amarillo blanco carne gris naranja negro rojo verde

headSet ("naranja"):  amarillo blanco carne gris
tailSet ("naranja"):  naranja negro rojo verde
primero: amarillo
ultimo : verde
```

Figura 19.19 | Uso de objetos `SortedSet` y `TreeSet`. (Parte 2 de 2).

El método `imprimirConjunto` (líneas 36 a 42) recibe un objeto `SortedSet` como argumento y lo imprime. En las líneas 38 y 39 imprime en pantalla cada elemento del objeto `SortedSet`, usando la instrucción `for` mejorada.

19.10 Mapas

Los objetos **`Map`** asocian claves a valores y no pueden contener claves duplicadas (es decir, cada clave puede asociarse solamente con un valor; a este tipo de asociación se le conoce como **asociación de uno a uno**. Los objetos `Map` difieren de los objetos `Set` en cuanto a que los primeros contienen claves y valores, mientras que los segundos contienen solamente valores. Tres de las muchas clases que implementan a la interfaz `Map` son **`HashTable`**, **`HashMap`** y **`TreeMap`**. Los objetos `HashTable` y `HashMap` almacenan elementos en tablas de hash, y los objetos `TreeMap` almacenan elementos en árboles. En esta sección veremos las tablas de hash y proporcionaremos un ejemplo en el que se utiliza un objeto `HashMap` para almacenar pares clavePvalor. La interfaz **`SortedMap`** extiende a `Map` y mantiene sus claves en orden; ya sea el orden natural de los elementos o un orden especificado por un objeto `Comparator`. La clase `TreeMap` implementa a `SortedMap`.

Implementación de `Map` con tablas de hash

Los lenguajes de programación orientados a objetos facilitan la creación de nuevos tipos. Cuando un programa crea objetos de tipos nuevos o existentes, es probable que necesite almacenarlos y obtenerlos con eficiencia. Los procesos de ordenar y obtener información con los arreglos es eficiente, si cierto aspecto de los datos coincide directamente con un valor de clave numérico, y si las claves son únicas y están estrechamente empaquetadas. Si tenemos 100 empleados con números de seguro social de nueve dígitos, y deseamos almacenar y obtener los datos de los empleados mediante el uso del número de seguro social como una clave, para ello requeriríamos un arreglo con mil millones de elementos, ya que hay mil millones de números únicos de nueve dígitos (000,000,000 a 999,999,999). Esto es impráctico para casi todas las aplicaciones que utilizan números de seguro social como claves. Un programa que tuviera un arreglo de ese tamaño podría lograr un alto rendimiento para almacenar y obtener registros de empleados, con sólo usar el número de seguro social como índice del arreglo.

Hay muchas aplicaciones con este problema; a saber, que las claves son del tipo incorrecto (por ejemplo, enteros no positivos que corresponden a los subíndices del arreglo) o que son del tipo correcto, pero se esparcen escasamente sobre un enorme rango. Lo que se necesita es un esquema de alta velocidad para convertir claves, como números de seguro social, números de piezas de inventario y demás, en índices únicos de arreglo. Así, cuando una aplicación necesite almacenar algo, el esquema podría convertir rápidamente la clave de la aplicación en un índice, y el registro podría almacenarse en esa posición en el arreglo. Para obtener datos se hace lo mismo: una vez que la aplicación tenga una clave para la que desee obtener un registro de datos, simplemente aplica la conversión a la clave; esto produce el índice del arreglo en el que se almacenan y obtienen los datos.

El esquema que describimos aquí es la base de una técnica conocida como **hashing**. ¿Por qué ese nombre? Al convertir una clave en un índice de arreglo, literalmente revolvemos los bits, formando un tipo de número "desordenado". En realidad, el número no tiene un significado real más allá de su utilidad para almacenar y obtener un registro de datos específico.

Un fallo en este esquema se denomina **colisión**; esto ocurre cuando dos claves distintas se asocian a la misma celda (o elemento) en el arreglo. No podemos almacenar dos valores en el mismo espacio, por lo que necesitamos encontrar un hogar alterno para todos los valores más allá del primero, que se asocie con un índice de arreglo específico. Hay muchos esquemas para hacer esto. Uno de ellos es "hacer hash de nuevo" (es decir, aplicar otra transformación de hashing a la clave, para proporcionar la siguiente celda como candidato en el arreglo). El pro-

ceso de hashing está diseñado para distribuir los valores en toda la tabla, por lo que se asume que se encontrará una celda disponible con sólo unas cuantas transformaciones de hashing.

Otro esquema utiliza un hash para localizar la primera celda candidata. Si esa celda está ocupada, se buscan celdas sucesivas en orden, hasta que se encuentra una disponible. El proceso de obtención funciona de la misma forma: se aplica hash a la clave una vez para determinar la función inicial y comprobar si contiene los datos deseados. Si es así, la búsqueda termina. En caso contrario, se busca linealmente en las celdas sucesivas hasta encontrar los datos deseados.

La solución más popular a las colisiones en las tablas de hash es hacer que cada celda de la tabla sea una "cubeta" de hash que, por lo general, viene siendo una lista enlazada de todos los pares clave/valor que se asocian con esa celda. Ésta es la solución que implementan las clases `Hashtable` y `HashMap` (del paquete `java.util`). Tanto `Hashtable` como `HashMap` implementan a la interfaz `Map`. Las principales diferencias entre ellas son que `HashMap` no está sincronizada (varios subprocesos no deben modificar un objeto `HashMap` en forma concurrente), y permite claves y valores `null`.

El **factor de carga** de una tabla de hash afecta al rendimiento de los esquemas de hashing. El factor de carga es la proporción del número de celdas ocupadas en la tabla de hash, con respecto al número total de celdas en la tabla de hash. Entre más se acerque esta proporción a 1.0, mayor será la probabilidad de colisiones.



Tip de rendimiento 19.7

El factor de carga en una tabla de hash es un clásico ejemplo de una concesión entre espacio de memoria y tiempo de ejecución: al incrementar el factor de carga, obtenemos un mejor uso de la memoria, pero el programa se ejecuta con más lentitud, debido al incremento en las colisiones de hashing. Al reducir el factor de carga, obtenemos más velocidad en la ejecución del programa, debido a la reducción en las colisiones de hashing, pero obtenemos un uso más pobre de la memoria, debido a que una proporción más grande de la tabla de hash permanece vacía.

Las tablas de hash son complejas de programar. Los estudiantes de ciencias computacionales estudian los esquemas de hashing en cursos titulados "Estructuras de datos" y "Algoritmos". Java proporciona las clases `Hashtable` y `HashMap` para permitir a los programadores utilizar la técnica de hashing sin tener que implementar los mecanismos de las tablas de hash. Este concepto es muy importante en nuestro estudio de la programación orientada a objetos. Como vimos en capítulos anteriores, las clases encapsulan y ocultan la complejidad (es decir, los detalles de implementación) y ofrecen interfaces amigables para el usuario. La fabricación apropiada de clases para exhibir tal comportamiento es una de las habilidades más valiosas en el campo de la programación orientada a objetos. En la figura 19.20 se utiliza un objeto `HashMap` para contar el número de ocurrencias de cada palabra en una cadena.

En la línea 17 se crea un objeto `HashMap` vacío con una capacidad inicial predeterminada (16 elementos) y un factor de carga predeterminado (0.75); estos valores predeterminados están integrados en la implementación de `HashMap`. Cuando el número de posiciones ocupadas en el objeto `HashMap` se vuelve mayor que la capacidad multiplicada por el factor de carga, la capacidad se duplica en forma automática. Observe que `HashMap` es una clase genérica que recibe dos argumentos de tipo. El primero especifica el tipo de clave (es decir, `String`) y el segundo el tipo de valor (es decir, `Integer`). Recuerde que los argumentos de tipo que se pasan a una clase gené-

```
1  // Fig. 19.20: ConteoTipoPalabras.java
2  // Programa que cuenta el número de ocurrencias de cada palabra en una cadena
3  import java.util.StringTokenizer;
4  import java.util.Map;
5  import java.util.HashMap;
6  import java.util.Set;
7  import java.util.TreeSet;
8  import java.util.Scanner;
9
10 public class ConteoTipoPalabras
11 {
12    private Map< String, Integer > mapa;
```

Figura 19.20 | Objetos `Hashmap` y `Map`. (Parte 1 de 3).

```
    private Scanner scanner;

    public ConteoTipoPalabras()
    {
       mapa = new HashMap< String, Integer >(); // crea objeto HashMap
       scanner = new Scanner( System.in ); // crea objeto scanner
       crearMap(); // crea un mapa con base en la entrada del usuario
       mostrarMap(); // muestra el contenido del mapa
    } // fin del constructor de ConteoTipoPalabras

    // crea mapa a partir de la entrada del usuario
    private void crearMap()
    {
       System.out.println( "Escriba una cadena:" ); // pide la entrada del usuario
       String entrada = scanner.nextLine();

       // crea objeto StringTokenizer para los datos de entrada
       StringTokenizer tokenizer = new StringTokenizer( entrada );

       // procesamiento del texto de entrada
       while ( tokenizer.hasMoreTokens() ) // mientras haya más entrada
       {
          String palabra = tokenizer.nextToken().toLowerCase(); // obtiene una palabra

          // si el mapa contiene la palabra
          if ( mapa.containsKey( palabra ) ) // está la palabra en el mapa?
          {
             int cuenta = mapa.get( palabra ); // obtiene la cuenta actual
             mapa.put( palabra, cuenta + 1 );  // incrementa la cuenta
          } // fin de if
          else
             mapa.put( palabra, 1 ); // agrega una nueva palabra con una cuenta de 1 al mapa
       } // fin de while
    } // fin del método crearMap

    // muestra el contenido del mapa
    private void mostrarMap()
    {
       Set< String > claves = mapa.keySet(); // obtiene las claves

       // ordena las claves
       TreeSet< String > clavesOrdenadas = new TreeSet< String >( claves );

       System.out.println( "El mapa contiene:\nClave\t\tValor" );

       // genera la salida para cada clave en el mapa
       for ( String clave : clavesOrdenadas )
          System.out.printf( "%-10s%10s\n", clave, mapa.get( clave ) );

       System.out.printf(
          "\nsize:%d\nisEmpty:%b\n", mapa.size(), mapa.isEmpty() );
    } // fin del método mostrarMap

    public static void main( String args[] )
    {
       new ConteoTipoPalabras();
    } // fin de main
} // fin de la clase ConteoTipoPalabras
```

Figura 19.20 | Objetos `Hashmap` y `Map`. (Parte 2 de 3).

```
Escriba una cadena:
Ser o no ser; esa es la pregunta Si es mas noble sufrir
El mapa contiene:
Clave          Valor
es                2
esa               1
la                1
mas               1
no                1
noble             1
o                 1
pregunta          1
ser               1
ser;              1
si                1
sufrir            1

size:12
isEmpty:false
```

Figura 19.20 | Objetos `Hashmap` y `Map`. (Parte 3 de 3).

rica deben ser tipos de referencias, por lo cual el segundo argumento de tipo es `Integer`, no `int`. En la línea 18 se crea un objeto `Scanner` que lee la entrada del usuario del flujo estándar de entrada. En la línea 19 se hace una llamada al método `crearMap` (líneas 24 a 46), el cual usa un `mapa` para almacenar el número de ocurrencias de cada palabra en la oración. En la línea 27 se invoca el método `nextLine` de `scanner` para obtener la entrada del usuario, y en la línea 30 se crea un objeto **`StringTokenizer`** para descomponer la cadena de entrada en sus palabras componentes individuales. Este constructor de `StringTokenizer` recibe un argumento de cadena y crea un objeto `StringTokenizer` para esa cadena, y utilizará el espacio en blanco para separarla. La condición en la instrucción `while` de las líneas 33 a 45 utiliza el método **`hasMoreTokens`** de `StringTokenizer` para determinar si hay más tokens en la cadena que se está separando en tokens. Si es así, en la línea 35 se convierte el siguiente token a minúsculas. El siguiente token se obtiene mediante una llamada al método **`nextToken`** de `StringTokenizer`, el cual devuelve un objeto `String`. [*Nota:* en la sección 30.6 hablaremos sobre la clase `StringTokenizer` con detalle]. Después, en la línea 38 se hace una llamada al método **`containsKey`** de `Mapa` para determinar si la palabra está en el mapa (y por ende, ha ocurrido antes en la cadena). Si el objeto `Mapa` no contiene una asignación para la palabra, en la línea 44 se utiliza el método **`put`** de `Mapa` para crear una nueva entrada en el mapa, con la palabra como la clave y un objeto `Integer` que contiene `1` como valor. Observe que la conversión autoboxing ocurre cuando el programa pasa el entero `1` al método `put`, ya que el mapa almacena el número de ocurrencias de la palabra como un objeto `Integer`. Si la palabra no existe en el mapa, en la línea 40 se utiliza el método **`get`** de `Mapa` para obtener el valor asociado de la clave (la cuenta) en el mapa. En la línea 41 se incrementa ese valor y se utiliza `put` para reemplazar el valor asociado de la clave en el mapa. El método `put` devuelve el valor anterior asociado con la clave, o `null` si la clave no estaba en el mapa.

El método `mostrarMap` (líneas 49 a 64) muestra todas las entradas en el mapa. Utiliza el método **`keySet`** (línea 51) de `HashMap` para obtener un conjunto de las claves. Estas claves tienen el tipo `String` en el `mapa`, por lo que el método `keySet` devuelve un tipo genérico `Set` con el parámetro de tipo especificado como `String`. En la línea 54 se crea un objeto `TreeSet` de las claves, en el cual se ordenan éstas. El ciclo en las líneas 59 a 60 accede a cada clave y a su valor en el mapa. En la línea 60 se muestra cada clave y su valor, usando el especificador de formato `%-10s` para justificar cada clave a la izquierda, y el especificador de formato `%10s` para justificar cada valor a la derecha. Observe que las claves se muestran en orden ascendente. En la línea 63 se hace una llamada al método **`size`** de `Mapa` para obtener el número de pares clave-valor en el objeto `Map`. En la línea 63 se hace una llamada a **`isEmpty`**, el cual devuelve un valor `boolean` que indica si el objeto `Map` está vacío o no.

19.11 La clase `Properties`

Un objeto **`Properties`** es un objeto `Hashtable` persistente que, por lo general, almacena pares clave-valor de cadenas; suponiendo que el programador utiliza los métodos **`setProperty`** y **`getProperty`** para manipular la

tabla, en vez de los métodos `put` y `get` heredados de `Hashtable`. Al decir "persistente", significa que el objeto `Properties` se puede escribir en un flujo de salida (posiblemente un archivo) y se puede leer de vuelta, a través de un flujo de entrada. Un uso común de los objetos `Properties` en versiones anteriores de Java era mantener los datos de configuración de una aplicación, o las preferencias del usuario para las aplicaciones. [*Nota:* la **API Preferences** (paquete **`java.util.prefs`**) está diseñada para reemplazar este uso específico de la clase `Properties`, pero esto se encuentra más allá del alcance de este libro. Para aprender más, visite el sitio Web `java.sun.com/javase/6/docs/technotes/guides/preferentes/index.html`].

La clase `Properties` extiende a la clase `Hashtable`. En la figura 19.21 se demuestran varios métodos de la clase `Properties`.

```
// Fig. 19.21: PruebaProperties.java
// Demuestra la clase Properties del paquete java.util.
import java.io.FileOutputStream;
import java.io.FileInputStream;
import java.io.IOException;
import java.util.Properties;
import java.util.Set;

public class PruebaProperties
{
   private Properties tabla;

   // establece la GUI para probar la tabla Properties
   public PruebaProperties()
   {
      tabla = new Properties(); // crea la tabla Properties

      // establece las propiedades
      tabla.setProperty( "color", "azul" );
      tabla.setProperty( "anchura", "200" );

      System.out.println( "Despues de establecer propiedades" );
      listarPropiedades(); // muestra los valores de las propiedades

      // reemplaza el valor de una propiedad
      tabla.setProperty( "color", "rojo" );

      System.out.println( "Despues de reemplazar propiedades" );
      listarPropiedades(); // muestra los valores de las propiedades

      guardarPropiedades(); // guarda las propiedades

      tabla.clear(); // vacia la tabla

      System.out.println( "Despues de borrar propiedades" );
      listarPropiedades(); // muestra los valores de las propiedades

      cargarPropiedades(); // carga las propiedades

      // obtiene el valor de la propiedad color
      Object valor = tabla.getProperty( "color" );

      // comprueba si el valor está en la tabla
      if ( valor != null )
         System.out.printf( "El valor de la propiedad color es %s\n", valor );
      else
         System.out.println( "La propiedad color no está en la tabla" );
```

Figura 19.21 | La clase `Properties` del paquete `java.util`. (Parte 1 de 3).

```
    } // fin del constructor de PruebaProperties

    // guarda las propiedades en un archivo
    public void guardarPropiedades()
    {
       // guarda el contenido de la tabla
       try
       {
          FileOutputStream salida = new FileOutputStream( "props.dat" );
          tabla.store( salida, "Propiedades de ejemplo" ); // guarda las propiedades
          salida.close();
          System.out.println( "Despues de guardar las propiedades" );
          listarPropiedades(); // muestra los valores de las propiedades
       } // fin de try
       catch ( IOException ioException )
       {
          ioException.printStackTrace();
       } // fin de catch
    } // fin del método guardarPropiedades

    // carga las propiedades de un archivo
    public void cargarPropiedades()
    {
       // carga el contenido de la tabla
       try
       {
          FileInputStream entrada = new FileInputStream( "props.dat" );
          tabla.load( entrada ); // carga las propiedades
          entrada.close();
          System.out.println( "Después de cargar las propiedades" );
          listarPropiedades(); // muestra los valores de las propiedades
       } // fin de try
       catch ( IOException ioException )
       {
          ioException.printStackTrace();
       } // fin de catch
    } // fin del método cargarPropiedades

    // imprime los valores de las propiedades
    public void listarPropiedades()
    {
       Set< Object > claves = tabla.keySet(); // obtiene los nombres de las propiedades

       // imprime los pares nombre/valor
       for ( Object clave : claves )
       {
          System.out.printf(
             "%s\t%s\n", clave, tabla.getProperty( ( String ) clave ) );
       } // fin de for

       System.out.println();
    } // fin del método listarPropiedades

    public static void main( String args[] )
    {
       new PruebaProperties();
    } // fin de main
 } // fin de la clase PruebaProperties
```

Figura 19.21 | La clase `Properties` del paquete `java.util`. (Parte 2 de 3).

```
Despues de establecer propiedades
anchura 200
color   azul

Despues de reemplazar propiedades
anchura 200
color   rojo

Despues de guardar las propiedades
anchura 200
color   rojo

Despues de borrar propiedades

Después de cargar las propiedades
anchura 200
color   rojo

El valor de la propiedad color es rojo
```

Figura 19.21 | La clase `Properties` del paquete `java.util`. (Parte 3 de 3).

En la línea 16 se utiliza el constructor sin argumentos para crear un objeto `Properties` llamado `tabla` sin propiedades predeterminadas. La clase `Properties` también cuenta con un constructor sobrecargado, el cual recibe una referencia a un objeto `Properties` que contiene valores de propiedad predeterminados. En cada una de las líneas 19 y 20 se hace una llamada al método `setProperty` de `Properties` para almacenar un valor para la clave especificada. Si la clave no existe en la `tabla`, `setProperty` devuelve `null`; en caso contrario, devuelve el valor anterior para esa clave.

En la línea 41 se llama al método `getProperty` de `Properties` para localizar el valor asociado con la clave especificada. Si la clave no se encuentra en este objeto `Properties`, `getProperty` devuelve `null`. Una versión sobrecargada de este método recibe un segundo argumento, el cual especifica el valor predeterminado a devolver si `getProperty` no puede localizar la clave.

En la línea 57 se hace una llamada al método **`store`** de `Properties` para guardar el contenido del objeto `Properties` en el objeto `OutputStream` especificado como el primer argumento (en este caso, el objeto `salida` de `FileOutputStream`). El segundo argumento, un objeto `String`, es una descripción del objeto `Properties`. La clase `Properties` también proporciona el método **`list`**, el cual recibe un argumento `PrintStream`. Este método es útil para mostrar la lista de propiedades.

En la línea 75 se hace una llamada al método **`load`** de `Properties` para restaurar el contenido del objeto `Properties` a partir del objeto `InputStream` especificado como el primer argumento (en este caso, un objeto `FileInputStream`). En la línea 89 se hace una llamada al método `keySet` de `Properties` para obtener un objeto `Set` de los nombres de las propiedades. En la línea 94 se obtiene el valor de una propiedad, para lo cual se pasa una clave al método `getProperty`.

19.12 Colecciones sincronizadas

En el capítulo 23 hablaremos sobre el subprocesamiento múltiple. Con la excepción de `Vector` y `Hashtable`, las colecciones en el marco de trabajo de colecciones están desincronizadas de manera predeterminada, por lo que pueden operar eficientemente cuando no se requiere el subprocesamiento múltiple. Sin embargo, debido a que están desincronizadas, el acceso concurrente a un objeto `Collection` por parte de varios subprocesos podría producir resultados indeterminados, o errores fatales. Para evitar potenciales problemas de subprocesamiento, se utilizan envolturas de sincronización para las colecciones que podrían ser utilizadas por varios subprocesos. Un objeto **envoltura** recibe llamadas a métodos, agrega la sincronización de subprocesos (para evitar un acceso concurrente a la colección) y delega las llamadas al objeto de la colección envuelto. La API `Collections` proporciona un conjunto de métodos `static` para envolver colecciones como versiones sincronizadas. En la figura 19.22 se enlistan los encabezados para las envolturas de sincronización. Los detalles acerca de estos métodos están dispo-

nibles en `java.sun.com/javase/6/docs/api/java/util/Collections.html`. Todos estos métodos reciben un tipo genérico como parámetro y devuelven una vista sincronizada del tipo genérico. Por ejemplo, el siguiente código crea un objeto `List` sincronizado (`lista2`) que almacena objetos `String`:

```
List< String > lista1 = new ArrayList< String >();
List< String > lista2 = Collections.synchronizedList( lista1 );
```

Encabezados de los métodos `public static`
`< T > Collection< T > synchronizedCollection( Collection< T > c )`
`< T > List< T > synchronizedList( List< T > unaLista )`
`< T > Set< T > synchronizedSet( Set< T > s )`
`< T > SortedSet< T > synchronizedSortedSet( SortedSet< T > s )`
`< K, V > Map< K, V > synchronizedMap( Map< K, V > m )`
`< K, V > SortedMap< K, V > synchronizedSortedMap( SortedMap< K, V > m )`

Figura 19.22 | Métodos de envoltura de sincronización.

19.13 Colecciones no modificables

La API `Collections` proporciona un conjunto de métodos `static` que crean **envolturas no modificables** para las colecciones. Las envolturas no modificables lanzan excepciones `UnsupportedOperationException` si se producen intentos por modificar la colección. En la figura 19.23 se enlistan los encabezados para estos métodos. Los detalles acerca de estos métodos están disponibles en `java.sun.com/javase/6/docs/api/java/util/Collections.html`. Todos estos métodos reciben un tipo genérico como parámetro y devuelven una vista no modificable del tipo genérico. Por ejemplo, el siguiente código crea un objeto `List` no modificable (`lista2`) que almacena objetos `String`:

```
List< String > lista1 = new ArrayList< String >();
List< String > lista2 = Collections.unmodifiableList( lista1 );
```



Observación de ingeniería de software 19.5

Puede utilizar una envoltura no modificable para crear una colección que ofrezca acceso de sólo lectura a otros, mientras que a usted le permita acceso de lectura/escritura. Para ello, simplemente dé a los otros una referencia a la envoltura no modificable, y usted conserve una referencia a la colección original.

Encabezados de los métodos `public static`
`< T > Collection< T > unmodifiableCollection( Collection< T > c )`
`< T > List< T > unmodifiableList( List< T > unaLista )`
`< T > Set< T > unmodifiableSet( Set< T > s )`
`< T > SortedSet< T > unmodifiableSortedSet( SortedSet< T > s )`
`< K, V > Map< K, V > unmodifiableMap( Map< K, V > m )`
`< K, V > SortedMap< K, V > unmodifiableSortedMap( SortedMap< K, V > m )`

Figura 19.23 | Métodos de envolturas no modificables.

19.14 Implementaciones abstractas

El marco de trabajo de colecciones proporciona varias implementaciones abstractas de interfaces de Collection, a partir de las cuales el programador puede construir implementaciones completas. Estas implementaciones abstractas incluyen una implementación de Collection delgada, llamada **AbstractCollection**; una implementación de List delgada, la cual permite el acceso aleatorio a sus elementos y se le conoce como **AbstractList**; una implementación de Map delgada conocida como **AbstractMap**, una implementación de List delgada que permite un acceso secuencial a sus elementos y se le conoce como **AbstractSequentialList**, una implementación de Set delgada, conocida como **AbstractSet**; y una implementación de Queue delgada, conocida como **AbstractQueue**. Puede aprender más acerca de estas clases en java.sun.com/javase/6/docs/api/java/util/package-summary.html.

Para escribir una implementación personalizada, puede extender la implementación abstracta que se adapte mejor a sus necesidades, e implementar cada uno de los métodos abstract de la clase. Después, si su colección es modificable, sobrescriba cualquier método concreto que evite su modificación.

19.15 Conclusión

En este capítulo se presentó el marco de tabajo de colecciones de Java. Aprendió a utilizar la clase Arrays para realizar manipulaciones con arreglos. Conoció la jerarquía de colecciones y aprendió a utilizar las interfaces del marco de trabajo de colecciones para programar con las colecciones mediante el polimorfismo. También conoció varios algoritmos predefinidos para manipular colecciones. En el siguiente capítulo presentaremos los applets de Java, los cuales son programas en Java que, por lo general, se ejecutan en un explorador Web. Empezaremos con applets de ejemplo que vienen con el JDK, y después le mostraremos cómo escribir y ejecutar sus propios applets.

Resumen

Sección 19.1 Introducción

- El marco de trabajo de colecciones de Java proporciona acceso al programador las estructuras de datos preempaquetadas, así como a los algoritmos para manipularlas.

Sección 19.2 Generalidades acerca de las colecciones

- Una colección es un objeto que puede contener referencias a otros objetos. Las interfaces de colecciones declaran las operaciones que pueden realizarse en cada tipo de colección.
- Las clases y las interfaces del marco de trabajo de colecciones se encuentran en el paquete java.util.

*Sección 19.3 La clase **Arrays***

- La clase Arrays proporciona métodos static para manipular arreglos, incluyendo a sort para ordenar un arreglo, a binarySearch para buscar en un arreglo ordenado, a equals para comparar arreglos y a fill para colocar elementos en un arreglo.
- El método asList de Arrays devuelve una vista List de un arreglo, la cual permite a un programa manipular el arreglo como si fuera un objeto List. Cualquier modificación realizada a través de la vista List modifica el arreglo, y cualquier modificación al arreglo modifica a la vista List.
- El método size obtiene el número de elementos en un objeto List, y el método get devuelve un elemento del objeto List.

*Sección 19.4 La interfaz **Collection** y la clase **Collections***

- La interfaz Collection es la interfaz raíz en la jerarquía de colecciones, a partir de la cual se derivan las interfaces Set y List. La interfaz Collection contiene operaciones masivas para agregar, borrar, comparar y retener objetos en una colección. La interfaz Collection proporciona un método llamado iterator para obtener un objeto Iterator.
- La clase Collections proporciona métodos static para manipular colecciones. Muchos de los métodos son implementaciones de algoritmos polimórficos para buscar, ordenar, etcétera.

Sección 19.5 Listas

- Un objeto `List` es un objeto `Collection` ordenado, que puede contener elementos duplicados.
- La interfaz `List` se implementa mediante las clases `ArrayList`, `LinkedList` y `Vector`. La clase `ArrayList` es una implementación tipo arreglo de un objeto `List`, que puede cambiar su tamaño. Un objeto `LinkedList` es una implementación tipo lista enlazada de un objeto `List`.
- El método `hasNext` de `Iterator` determina si un objeto `Collection` contiene otro elemento. El método `next` devuelve una referencia al siguiente objeto en el objeto `Collection`, y avanza el objeto `Iterator`.
- El método `subList` devuelve una vista de una porción de un objeto `List`. Cualquier modificación realizada en esta vista se realiza también en el objeto `List`.
- El método `clear` elimina elementos de un objeto `List`.
- El método `toArray` devuelve el contenido de una colección, en forma de un arreglo.
- La clase `Vector` maneja arreglos que pueden cambiar su tamaño en forma dinámica. En cualquier momento dado, un objeto `Vector` contiene un número de elementos menor o igual a su capacidad. Si un objeto `Vector` necesita crecer, aumenta en base a su incremento de capacidad. Si no se especifica un incremento de capacidad, Java duplica el tamaño del objeto `Vector` cada vez que se requiere una capacidad adicional. La capacidad predeterminada es de 10 elementos.
- El método `add` de `Vector` agrega su argumento al final del objeto `Vector`. El método `insertElementAt` inserta un elemento en la posición especificada. El método `set` establece el elemento en una posición específica.
- El método `remove` de `Vector` elimina del objeto `Vector` la primera ocurrencia de su argumento. El método `removeAllElements` elimina todos los elementos del objeto `Vector` El método `removeElementAt` elimina el elemento en el índice especificado.
- El método `firstElement` de `Vector` devuelve una referencia al primer elemento. El método `lastElement` devuelve una referencia al último elemento.
- El método `contains` de `Vector` determina si el objeto `Vector` contiene la `claveBusqueda` especificada como argumento. El método `indexOf` de `Vector` obtiene el índice de la primera ubicación de su argumento. El método devuelve `-1` si el argumento no se encuentra en el objeto `Vector`.
- El método `isEmpty` de `Vector` determina si el objeto `Vector` está vacío. Los métodos `size` y `capacity` determinan el número de elementos actuales en el objeto `Vector`, y el número de elementos que pueden almacenarse en el objeto `Vector` sin asignar más memoria, respectivamente.

Sección 19.6 Algoritmos de colecciones

- Los algoritmos `sort`, `binarySearch`, `reverse`, `shuffle`, `fill` y `copy` operan en objetos `List`. Los algoritmos `min` y `max` operan en objetos `Collection`. El algoritmo `reverse` invierte los elementos de un objeto `List`, el algoritmo `fill` establece cada elemento del objeto `List` a un objeto `Object` especificado, y `copy` copia elementos de un objeto `List` a otro objeto `List`. El algoritmo `sort` ordena los elementos de un objeto `List`.
- El algoritmo `addAll` anexa a una colección todos los elementos en un arreglo, el algoritmo `frequency` calcula cuántos elementos en la colección son iguales al elemento especificado, y `disjoint` determina si dos colecciones tienen elementos en común.
- Los algoritmos `min` y `max` buscan los elementos mayor y menor en una colección.
- La interfaz `Comparator` proporciona un medio para ordenar los elementos de un objeto `Collection` en un orden distinto a su orden natural.
- El método `reverseOrder` de `Collections` devuelve un objeto `Comparator` que puede usarse con `sort` para ordenar elementos de una colección en forma inversa.
- El algoritmo `shuffle` ordena al azar los elementos de un objeto `List`.
- El algoritmo `binarySearch` localiza un objeto `Object` en un objeto `List` ordenado.

Sección 19.7 La clase `Stack` *del paquete* `java.util`

- La clase `Stack` extiende a `Vector`. El método `push` de `Stack` agrega su argumento a la parte superior de la pila. El método `pop` elimina el elemento superior de la pila. El método `peek` devuelve una referencia al elemento superior sin eliminarlo. El método `empty` de `Stack` determina si la pila está vacía o no.

Sección 19.8 La clase `PriorityQueue` *y la interfaz* `Queue`

- `Queue`, una nueva interfaz de colecciones presentada en Java SE 5, extiende a la interfaz `Collection` y proporciona operaciones adicionales para insertar, eliminar e inspeccionar elementos en una cola.
- `PriorityQueue`, una de las implementaciones de `Queue`, ordena los elementos en base a su orden natural (es decir, la implementación del método `compareTo`) o mediante un objeto `Comparator` que se suministra a través del constructor.

- Las operaciones comunes de `PriorityQueue` son: `offer` para insertar un elemento en la ubicación apropiada, con base en el orden de prioridad; `poll` para eliminar el elemento de mayor prioridad de la cola de prioridad (es decir, la parte inicial o cabeza de la cola); `peek` para obtener una referencia al elemento de mayor prioridad de la cola de prioridad; `clear` para eliminar todos los elementos de la cola de prioridad; y `size` para obtener el número de elementos en la cola de prioridad.

Sección 19.9 Conjuntos
- Un objeto `Set` es un objeto `Collection` que no contiene elementos duplicados. `HashSet` almacena sus elementos en una tabla de hash. `TreeSet` almacena sus elementos en un árbol.
- La interfaz `SortedSet` extiende a `Set` y representa un conjunto que mantiene sus elementos ordenados. La clase `TreeSet` implementa a `SortedSet`.
- El método `headSet` de `TreeSet` obtiene una vista de un objeto `TreeSet` que es menor a un elemento especificado. El método `tailSet` obtiene una vista que es mayor o igual a un elemento especificado. Cualquier modificación realizada a la vista se realiza al objeto `TreeSet`.

Sección 19.10 Mapas
- Los objetos `Map` asocian claves con valores y no pueden contener claves duplicadas. Los objetos `Map` difieren de los objetos `Set` en cuanto a que los objetos `Map` contienen tanto claves como valores, mientras que los objetos `Set` sólo contienen valores. Los objetos `HashMap` almacenan elementos en una tabla de hash, y los objetos `TreeMap` almacenan elementos en un árbol.
- Los objetos `Hashtable` y `HashMap` almacenan elementos en tablas de hash, y los objetos `TreeMap` almacenan elementos en árboles.
- `HashMap` es una clase genérica que recibe dos argumentos de tipo. El primer argumento de tipo especifica el tipo de la clave, y el segundo especifica el tipo de valor.
- El método `put` de `HashMap` agrega una clave y un valor en un objeto `HashMap`. El método `get` localiza el valor asociado con la clave especificada. El método `isEmpty` determina si el mapa está vacío.
- El método `keySet` de `HashMap` devuelve un conjunto de las claves. Los métodos `size` e `isEmpty` de `map` devuelven el número de pares clave-valor en el objeto `Map`, y un valor booleano que indica si el objeto `Map` está vacío, respectivamente.
- La interfaz `SortedMap` extiende a `Map` y representa un mapa que mantiene sus claves en orden. La clase `TreeMap` implementa a `SortedMap`.

Sección 19.11 La clase `Properties`
- Un objeto `Properties` es un objeto `Hashtable` persistente. La clase `Properties` extiende a `Hashtable`.
- El constructor de `Properties` sin argumentos crea una tabla `Properties` vacía sin propiedades predeterminadas. También hay un constructor sobrecargado que recibe una referencia a un objeto `Properties` predeterminado que contiene valores de propiedades predeterminados.
- El método `setProperty` de `Properties` especifica el valor asociado con el argumento tipo clave. El método `getProperty` de `Properties` localiza el valor de la clave especificada como argumento. El método `store` guarda el contenido del objeto `Properties` en el objeto `OutputStream` especificado como el primer argumento. El método `load` restaura el contenido del objeto `Properties` del objeto `InputStream` que se especifica como el argumento.

Sección 19.12 Colecciones sincronizadas
- Las colecciones del marco de trabajo de colecciones están desincronizadas. Las envolturas de sincronización se proporcionan para las colecciones a las que pueden acceder varios subprocesos en forma simultánea.

Sección 19.13 Colecciones no modificables
- La API `Collections` proporciona un conjunto de métodos `public static` para convertir colecciones en versiones no modificables. Las envolturas no modificables lanzan excepciones `UnsupportedOperationException` si hay intentos de modificar la colección.

Sección 19.14 Implementaciones abstractas
- El marco de trabajo de colecciones proporciona varias implementaciones abstractas de las interfaces de colecciones, a partir de las cuales el programador puede crear rápidamente implementaciones personalizadas completas.

Terminología

AbstractCollection, clase
AbstractList, clase
AbstractMap, clase
AbstractQueue, clase
AbstractSequentialList, clase
AbstractSet, clase
add, método de List
add, método de Vector
addAll, método de Collections
addFirst, método de List
addLast, método de List
algoritmos en Collections
ArrayList
arreglo
arreglos como colecciones
asignación de uno a uno
asignaciones
asList, método de Arrays
asociar claves con valores
binarySearch, método de Arrays
binarySearch, método de Collections
capacity, método de Vector
clase de envoltura
clave en HashMap
clear, método de List
clear, método de PriorityQueue
colección ordenada
colecciones colocadas en arreglos
colecciones modificables
colecciones no modificables
colisión en hashing
collection
Collection, interfaz
Collections, clase
Comparable, interfaz
comparación lexicográfica
Comparator, interfaz
compareTo, método de Comparable
contains, método de vector
containsKey, método de HashMap
copy, método de Collections
disjoint, método de Collections
elementos duplicados
eliminar un elemento de una colección
envolturas de sincronización
factor de carga en hashing
fill, método de Arrays
fill, método de Collections
firstElement, método de Vector
frequency, método de Collections
get, método de HashMap
getProperty, método de la clase Properties
hashing
HashMap, clase
HashSet, clase
Hashtable, clase
hasMoreTokens, método de StringTokenizer
hasNext, método de Iterator
hasPrevious, método de ListIterator
incremento de capacidad de un objeto Vector
indexOf, método de Vector
insertar un elemento en una colección
isEmpty, método de Map
isEmpty, método de Vector
iterador
iterador bidireccional
iterar a través de los elementos de un contenedor
Iterator, interfaz
keySet, método de HashMap
lastElement, método de Vector
LinkedList, clase
List, interfaz
ListIterator, interfaz
Map, interfaz de colección
mapa
marco de trabajo de colecciones
max, método de Collections
método de comparación natural
métodos de vista de rango
min, método de Collections
next, método de Iterator
nextToken, método de StringTokenizer
NoSuchElementException, clase
offer, método de PriorityQueue
ordenamiento
ordenamiento estable
ordenamiento natural
ordenar un objeto List
par clave/valor
peek, método de PriorityQueue
peek, método de Stack
poll, método de PriorityQueue
pop, método de Stack
PriorityQueue, clase
Properties, clase
put, método de HashMap
Queue, interfaz
removeAllElements, método de Vector
removeElement, método de Vector
removeElementAt, método de Vector
reverse, método de Collections
reverseOrder, método de Collections
secuencia
Set, interfaz
shuffle, método de Collections
size, método de List
size, método de PriorityQueue
sort, método de Arrays
sort, método de Collections
SortedMap, interfaz de colección

SortedSet, interfaz de colección
Stack, clase
StringTokenizer, clase
TreeMap, clase
TreeSet, clase
ver un arreglo como un objeto List
vista

Ejercicios de autoevaluación

19.1 Complete las siguientes oraciones:
a) Un(a) ____________ se utiliza para recorrer una colección y puede eliminar elementos de la colección, durante la iteración.
b) Para acceder a un elemento en un objeto List, se utiliza el ____________ del elemento.
c) A los objetos List se les conoce algunas veces como ____________.
d) Las clases ____________ y ____________ de Java proporcionan las herramientas de estructuras de datos tipo arreglo, que pueden cambiar su tamaño en forma dinámica.
e) Si usted no especifica un incremento de capacidad, el sistema ____________ el tamaño del objeto Vector cada vez que se requiere una capacidad adicional.
f) Puede utilizar un(a) ____________ para crear una colección que ofrezca acceso de sólo lectura a los demás, mientras que a usted le permita el acceso de lectura/escritura.
g) Los objetos ____________ se pueden utilizar para crear pilas, colas, árboles y deques (colas con doble extremo).
h) El algoritmo ____________ de Collections determina si dos colecciones tienen elementos en común.

19.2 Conteste con *verdadero* o *falso* a cada una de las siguientes proposiciones; en caso de ser *falso*, explique por qué.
a) Los valores de tipos primitivos pueden almacenarse directamente en un objeto Vector.
b) Un objeto Set puede contener valores duplicados.
c) Un objeto Map puede contener claves duplicadas.
d) Un objeto LinkedList puede contener valores duplicados.
e) Collections es una interfaz (interface).
f) Los objetos Iterator pueden eliminar elementos.
g) Con la técnica de hashing, a medida que se incrementa el factor de carga, disminuye la probabilidad de colisiones.
h) Un objeto PriorityQueue permite elementos null.

Respuestas a los ejercicios de autoevaluación

19.1 a) Iterator. b) índice. c) secuencias. d) ArrayList, Vector. e) duplicará. f) no modificable wrapper. g) LinkedLists. h) disjoint.

19.2 a) Falso; un objeto Vector sólo almacena objetos. La conversión autoboxing ocurre cuando se agrega un tipo primitivo al objeto Vector, lo cual significa que el tipo primitivo se convierte en su clase de envoltura de tipo correspondiente.
b) Falso. Un objeto Set no puede contener valores duplicados.
c) Falso. Un objeto Map no puede contener claves duplicadas.
d) Verdadero.
e) Falso. Collections es una clase; Collection es una interfaz (interface).
f) Verdadero.
g) Falso. Con la técnica de hashing, a medida que aumenta el factor de carga, hay menos posiciones disponibles, relativas al número total de posiciones, por lo que la probabilidad de seleccionar una posición ocupada (una colisión) con una operación de hashing se incrementa.
h) Falso. Una excepción NullPointerException se lanza si el programa trata de agregar null a un objeto PriorityQueue.

Ejercicios

19.3 Defina cada uno de los siguientes términos:
a) Collection
b) Collections

c) `Comparator`
d) `List`
e) factor de carga
f) colisión
g) concesión entre espacio y tiempo en hashing
h) `HashMap`

19.4 Explique brevemente la operación de cada uno de los siguientes métodos de la clase `Vector`:
a) `add`
b) `insertElementAt`
c) `set`
d) `remove`
e) `removeAllElements`
f) `removeElementAt`
g) `firstElement`
h) `lastElement`
i) `isEmpty`
j) `contains`
k) `indexOf`
l) `size`
m) `capacity`

19.5 Explique por qué la operación de insertar elementos adicionales en un objeto `Vector`, cuyo tamaño actual sea menor que su capacidad, es una operación relativamente rápida, y por qué el insertar elementos adicionales en un objeto `Vector`, cuyo tamaño actual sea igual a la capacidad, es una operación relativamente baja.

19.6 Al extender la clase `Vector`, los diseñadores de Java pudieron crear la clase `Stack` rápidamente. ¿Cuáles son los aspectos negativos de este uso de la herencia, en especial para la clase `Stack`?

19.7 Responda brevemente a las siguientes preguntas:
a) ¿Cuál es la principal diferencia entre un objeto `Set` y un objeto `Map`?
b) ¿Puede pasarse un arreglo bidimensional al método `asList` de `Arrays`? Si es así, ¿cómo se accedería a un elemento individual?
c) ¿Qué ocurre cuando agregamos un valor de tipo primitivo (por ejemplo, `double`) a una colección?
d) ¿Podemos imprimir todos los elementos en una colección sin utilizar un objeto `Iterator`? Si es así, explique cómo.

19.8 Explique brevemente la operación de cada uno de los siguientes métodos relacionados con `Iterator`:
a) `iterator`
b) `hasNext`
c) `next`

19.9 Explique brevemente la operación de cada uno de los siguientes métodos de la clase `HashMap`:
a) `put`
b) `get`
c) `isEmpty`
d) `containsKey`
e) `keySet`

19.10 Determine si cada uno de los siguientes enunciados es *verdadero* o *falso*. Si es *falso*, explique por qué.
a) Los elementos en un objeto `Collection` deben almacenarse en orden ascendente, antes de poder realizar una búsqueda binaria mediante `binarySearch`.
b) El método `first` obtiene el primer elemento en un objeto `TreeSet`.
c) Un objeto `List` creado con el método `asList` de `Arrays` puede cambiar su tamaño.
d) La clase `Arrays` proporciona el método `static` llamado `sort` para ordenar los elementos de un arreglo.

19.11 Explique la operación de cada uno de los siguientes métodos de la clase `Properties`:
a) `load`
b) `store`

c) `getProperty`
d) `list`

19.12 Vuelva a escribir las líneas 17 a 26 en la figura 19.4 para que sean más concisas; utilice el método `asList` y el constructor de `LinkedList` que recibe un argumento `Collection`.

19.13 Escriba un programa que lea una serie de nombres de pila y los almacene en un objeto `LinkedList`. No almacene nombres duplicados. Permita al usuario buscar un nombre de pila.

19.14 Modifique el programa de la figura 19.20 para contar el número de ocurrencias de cada letra, en vez de cada palabra. Por ejemplo, la cadena `"HOLA A TODOS"` contiene una `H`, tres `Os`, una `L`, dos `As`, una `T`, una `D` y una `S`. Muestre los resultados.

19.15 Use un objeto `HashMap` para crear una clase reutilizable y elegir uno de los 13 colores predefinidos en la clase `Color`. Los nombres de los colores deben usarse como claves, y los objetos `Color` predefinidos deben usarse como valores. Coloque esta clase en un paquete que pueda importarse en cualquier programa en Java. Use su nueva clase en una aplicación que permita al usuario seleccionar un color y dibujar una figura en ese color.

19.16 Escriba un programa que determine e imprima el número de palabras duplicadas en un enunciado. Trate a las letras mayúsculas y minúsculas de igual forma. Ignore los signos de puntuación.

19.17 Vuelva a escribir su solución al ejercicio 17.8 para utilizar una colección `LinkedList`.

19.18 Vuelva a escribir su solución al ejercicio 17.9 para utilizar una colección `LinkedList`.

19.19 Escriba un programa que reciba una entrada tipo número entero de un usuario, y que determine si es primo. Si el número no es primo, muestre sus factores primos únicos. Recuerde que los factores de un número primo son sólo 1 y el mismo número primo. Todo número que no sea primo tiene una factorización prima única. Por ejemplo, considere el número 54. Los factores primos de 54 son 2, 3, 3 y 3. Cuando los valores se multiplican entre sí, el resultado es 54. Para el número 54, los factores primos a imprimir deben ser 2 y 3. Use objetos `Set` como parte de su solución.

19.20 Escriba un programa que utilice un objeto `StringTokenizer` para dividir en tokens una línea de texto introducida por el usuario, y que coloque cada token en un objeto `TreeSet`. Imprima los elementos del objeto `TreeSet`. [*Nota:* esto debe hacer que se impriman los elementos en orden ascendente].

19.21 Los resultados de la figura 19.17 (`PriorityQueueTest`) muestra que `PriorityQueue` ordena elementos `Double` en orden ascendente. Vuelva a escribir la figura 19.17, de manera que ordene los elementos `Double` en orden descendente (es decir, `9.8` debe ser el elemento de mayor prioridad, en vez de `3.2`).

20

Introducción a los applets de Java



Observe las medidas adecuadas, ya que de todas las cosas, la sincronización correcta es el factor más importante.

— Hesiod

La pintura es el puente que vincula la mente del pintor con la del observador.

— Eugene Delacroix

La dirección en la que la educación empiece a guiar a un hombre determinará su futuro en la vida.

— Platón

OBJETIVOS

En este capítulo aprenderá a:

- Diferenciar entre applets (subprogramas) y aplicaciones.
- Observar algunas de las excitantes características de Java a través de los applets de demostración incluidos en el JDK.
- Escribir applets simples en Java.
- Escribir un documento HTML (HyperText Markup Language, Lenguaje de Marcado de Hipertexto) para cargar un applet en un contenedor de applets y ejecutarlo.
- Utilizar cinco métodos que el contenedor de un applet llama de manera automática durante el ciclo de vida del applet.

Plan general

20.1 Introducción

[*Nota:* este capítulo y sus ejercicios son pequeños y simples de manera intencional, para los lectores que desean aprender acerca de los applets después de leer sólo los primeros capítulos del libro; posiblemente los capítulos 2 y 3. En el capítulo 21, Multimedia: Applets y aplicaciones, en el capítulo 23, Subprocesamiento múltiple, y en el capítulo 24, Redes, presentaremos applets más complejos].

En este capítulo se introducen los **applets**: programas en Java que pueden incrustarse en **documentos HTML (Lenguaje de marcado de hipertexto)** (es decir, páginas Web). Cuando un explorador carga una página Web que contiene un applet, éste se descarga en el explorador Web y se ejecuta.

Al explorador que ejecuta un applet se le conoce como **contenedor de applets**. El JDK incluye el contenedor de applets **`appletviewer`** para probar applets a medida que se van desarrollando, y antes de incrustarlas en las páginas Web. Por lo general, demostraremos los applets mediante el uso del `appletviewer`. Si desea ejecutar sus applets en un explorador Web, debe estar consciente de que algunos exploradores Web no soportan a Java de manera predeterminada. Puede visitar `java.com` y hacer clic en el botón **Descargar AHORA** para instalar Java en su explorador Web. Hay soporte para varios exploradores Web populares.

20.2 Applets de muestra incluidos en el JDK

Comencemos por considerar varios applets de muestra que se incluyen con el JDK. Cada applet de muestra incluye su código fuente. Algunos programadores encuentran interesante leer este código fuente para aprender nuevas y excitantes características sobre Java. demuestran una pequeña porción de las poderosas herramientas de Java.

Los programas de demostración que se proporcionan con el JDK se encuentran en un directorio llamado `demo`. Para Windows, la ubicación predeterminada del directorio `demo` del JDK 6.0 es

```
C:\Archivos de programa\Java\jdk1.6.0\demo
```

En UNIX/Linux/Mac OS X, la ubicación predeterminada es el directorio en el que usted haya instalado el JDK, seguido de `jdk1.6.0/demo`. Por ejemplo,

```
/usr/local/jdk1.6.0/demo
```

En las demás plataformas hay una estructura similar de directorios (o carpetas). Este capítulo supone que el JDK está instalado en `C:\Archivos de programa\Java\jdk1.6.0_01\demo` en Windows, y en su directorio personal `~/jdk1.6.0` en UNIX/Linux/Mac OS X. Tal vez necesite actualizar las ubicaciones que se especifican aquí para reflejar el directorio de instalación y la unidad de disco que usted eligió, o una versión distinta del JDK.

Si utiliza una herramienta de desarrollo de Java que no incluya los programas de muestra de Sun Java, puede descargar el JDK (con los demos) en el sitio Web de Java de Sun Microsystems

```
java.sun.com/javase/6/
```

Applet `TresEnRaya`

El applet de demostración `TresEnRaya` (también conocido como gato) le permite a usted jugar contra la computadora. Para ejecutar este applet, abra una ventana de comandos y vaya al directorio `demo` del JDK.

El directorio `demo` contiene varios subdirectorios. Puede ver estos directorios escribiendo el comando `dir` en la ventana de comandos en Windows, o el comando `ls` en UNIX/Linux/Mac OS X. Hablaremos sobre los programas de muestra en los directorios `applets` y `jfc`. El directorio `applets` contiene varios applets de demostración. El directorio `jfc` (Java Foundation Classes, Clases Fundamentales de Java) contiene applets y aplicaciones que demuestran las características de gráficos y GUI de Java.

Cambie al directorio `applets` y muestre su contenido para ver los nombres de los directorios para los applets de demostración. En la figura 20.1 se proporciona una breve descripción de cada applet de muestra. Si su explorador Web soporta Java, puede probar estos applets abriendo el sitio Web `java.sun.com/javase/6/docs/technotesamples/demos.html` en su explorador y haciendo clic en el vínculo **Applets Page**. Demostraremos tres de estos applets usando el comando `appletviewer` en una ventana de comandos.

Ejemplo	Descripción
`Animator`	Realiza una de cuatro animaciones separadas.
`ArcTest`	Demuestra cómo dibujar arcos. Puede interactuar con el applet para modificar los atributos del arco que se muestra en pantalla.
`BarChart`	Dibuja un gráfico de barras simple.
`Blink`	Muestra texto destellante en distintos colores.
`CardTest`	Demuestra varios componentes y esquemas de la GUI.
`Clock`	Dibuja un reloj con manecillas giratorias, la fecha actual y la ahora actual. El reloj se actualiza una vez por segundo.
`DitherTest`	Demuestra cómo dibujar con una técnica de gráficos conocida como difuminado, la cual permite una transformación gradual de un color a otro.
`DrawTest`	Permite usar el ratón para dibujar líneas y puntos en distintos colores, arrastrando el ratón.
`Fractal`	Dibuja un fractal. Por lo general, los fractales requieren cálculos complejos para determinar la forma en que se muestran en la pantalla.
`GraphicsTest`	Dibuja figuras para ilustrar las herramientas de gráficos.
`GraphLayout`	Dibuja un gráfico que consiste en muchos nodos (representados como rectángulos) conectados por líneas. Arrastre un nodo para ver cómo se ajustan los demás nodos en el gráfico en la pantalla, y demostrar las interacciones gráficas complejas.
`ImageMap`	Demuestra una imagen con puntos activos. Al posicionar el puntero del ratón sobre ciertas áreas de la imagen se resalta el área y se muestra un mensaje en la esquina inferior izquierda de la ventana del contenedor de applets. Colóquese sobre la boca para escuchar cómo la imagen dice “hola.”
`JumpingBox`	Desplaza un rectángulo en forma aleatoria, alrededor de la pantalla. ¡Trate de atraparlo haciendo clic sobre él con el ratón!
`MoleculeViewer`	Presenta una vista tridimensional de varias moléculas químicas distintas. Arrastre el ratón y verá la molécula desde varios ángulos.
`NervousText`	Arrastra texto que salta por la pantalla.
`SimpleGraph`	Dibuja una curva compleja.

Figura 20.1 | Los ejemplos del directorio `applets`. (Parte 1 de 2).

Ejemplo	Descripción
`SortDemo`	Compara tres técnicas de ordenamiento. El ordenamiento (que se describe en el capítulo 16) sirve para organizar la información; es como alfabetizar las palabras. Cuando usted ejecuta este ejemplo desde una ventana de comandos, aparecen tres ventanas del appletviewer. Cuando ejecuta este ejemplo en un explorador Web, los tres ejemplos aparecen uno al lado del otro. Haga clic en cada una de ellas para empezar con el ordenamiento. Observe que cada una de las tres técnicas de ordenamiento operan a distintas velocidades.
`SpreadSheet`	Muestra una hoja de cálculo simple, con filas y columnas.
`TicTacToe`	Permite al usuario jugar al Tres en raya contra la computadora.
`WireFrame`	Dibuja una figura tridimensional como una malla de alambre. Arrastre el ratón para ver la figura desde distintos ángulos.

Figura 20.1 | Los ejemplos del directorio `applets`. (Parte 2 de 2).

Cambie al subdirectorio `TicTacToe`, en donde encontrará el documento HTML `example1.html` que se utiliza para ejecutar el applet. En la ventana de comandos, escriba el comando

```
appletviewer example1.html
```

y oprima la tecla *Entrar*. Esto hace que se ejecute el contenedor de applets `appletviewer`, el cual carga el documento HTML `example1.html` que se especifica como su argumento de línea de comandos. El `appletviewer` determina en base al documento qué applet debe cargar y comienza a ejecutarlo. La figura 20.2 muestra varias capturas de pantalla del juego de Tres en raya con este applet.

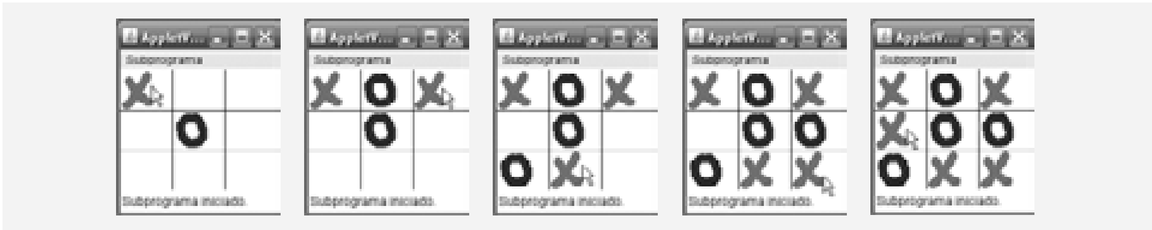


Figura 20.2 | Ejecución de ejemplo del applet Tres en raya.

Usted es el jugador **X**. Para interactuar con el applet, coloque el ratón sobre el cuadro en el que desea colocar una **X** y haga clic con el botón del ratón. El applet reproduce un sonido y coloca una **X** en el cuadro, si éste está libre. Si el cuadro está ocupado, es un movimiento inválido y el applet reproduce un sonido distinto, indicando que usted no puede hacer el movimiento especificado. Después de que haga un movimiento válido, el applet responderá con su propio movimiento.

Para jugar de nuevo, haga clic en el **menú Subprograma (Applet)** del `appletviewer` y seleccione el **elemento de menú Volver a cargar (Reload)** (Figura 20.3). Para terminar el `appletviewer`, haga clic en el menú **Subprograma** y seleccione el **elemento de menú Salir (Quit)**.

Applet `DrawTest`

El applet de demostración `DrawTest` le permite dibujar líneas y puntos en distintos colores. En la ventana de comandos, cambie al directorio `applets` y después al subdirectorio `DrawTest`. Puede desplazarse hacia arriba del árbol de directorios para llegar a `demo`, mediante el comando "`cd ..`" en la ventana de comandos. El directorio `DrawTest` contiene el documento `example1.html` que se utiliza para ejecutar el applet. En la ventana de comandos, escriba el comando

```
appletviewer example1.html
```

y oprima la tecla *Entrar*. El `appletviewer` carga `example1.html`, determina en base a este archivo qué applet cargar y comienza a ejecutarlo. La figura 20.4 muestra una captura de pantalla de este applet, después de dibujar algunas líneas y puntos.

De manera predeterminada, el applet nos permite dibujar líneas de color negro, arrastrando el ratón a lo largo del applet. Al arrastrar el ratón, observe que el punto inicial de la línea siempre permanece en el mismo lugar, y el punto final de la línea sigue al ratón a lo largo del applet. La línea no es permanente sino hasta que se libera el botón del ratón.

Para seleccionar un color, haga clic en uno de los botones de opción en la parte inferior del applet. Puede seleccionar de entre rojo, verde, azul, rosa, naranja y negro. Cambie la figura a dibujar de líneas (**Lines**) a (**Points**) al seleccionar **Points** en el cuadro combinado. Para empezar un nuevo dibujo, seleccione **Volver a cargar** en el menú **Subprograma** del `appletviewer`.



Figura 20.3 | Menú **applet** en el `appletviewer`.

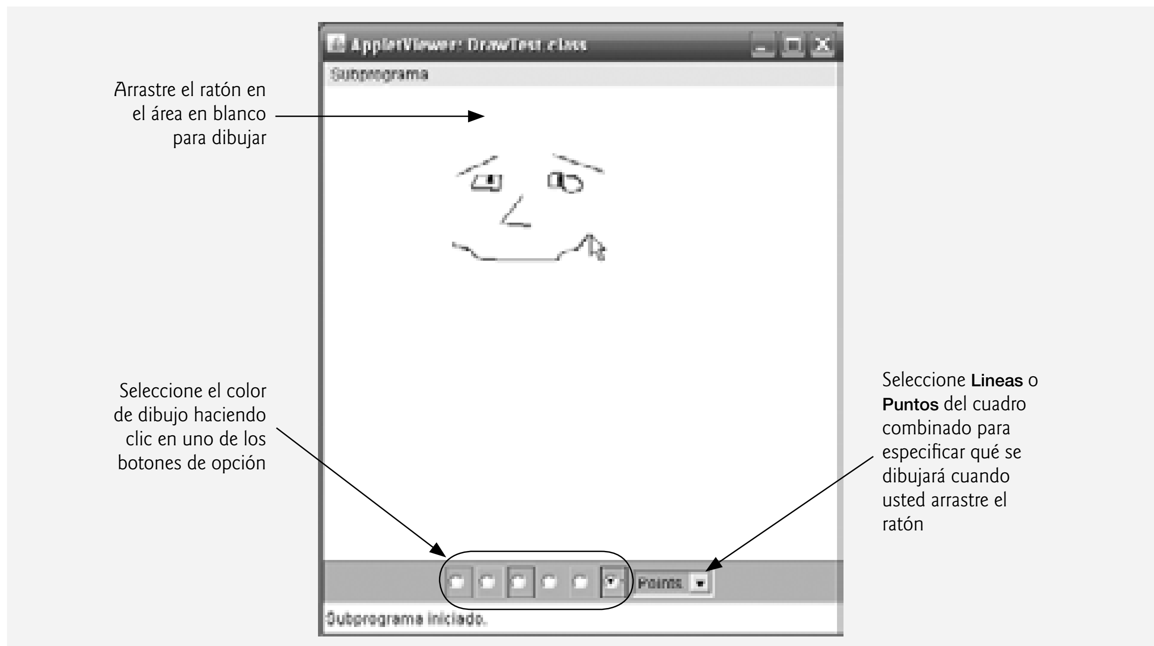


Figura 20.4 | Ejecución de ejemplo del applet `DrawTest`.

Applet `Java2D`

Este applet demuestra muchas características de la API Java2D (que presentamos en el capítulo 12). Cambie al directorio `jfc` que se encuentra en el directorio `demo` del JDK, y después cambie al directorio `Java2D`. En la ventana de comandos, escriba el comando

```
appletviewer Java2Demo.html
```

y oprima *Intro*. El `appletviewer` carga `Java2Demo.html`, determina en base al documento qué applet debe cargar y comienza a ejecutarlo. En la figura 20.5 se muestra una captura de pantalla de una de las muchas demostraciones de este applet, en relación con las herramientas para gráficos bidimensionales de Java.

En la parte superior del applet hay fichas que parecen carpetas en un archivero. Esta demostración cuenta con 12 fichas, en cada una de las cuales se demuestran las características de la API Java 2D. Para cambiar a una parte distinta de la demostración, simplemente haga clic en otra ficha. También puede probar cambiando las opciones en la esquina superior derecha del applet. Algunas de estas opciones afectan la velocidad con la que el applet dibuja los gráficos. Por ejemplo, haga clic en la casilla de verificación que está a la izquierda de la palabra **Anti-Aliasing** para activar y desactivar el suavizado (una técnica de gráficos para producir gráficos en pantalla más suaves, en los que los bordes del gráfico están desenfocados). Cuando esta característica se desactiva, la velocidad de animación aumenta para las figuras animadas que están en la parte inferior de la demostración (figura 20.5). Este incremento en el rendimiento ocurre debido a que las figuras que no están suavizadas son menos complejas de dibujar.

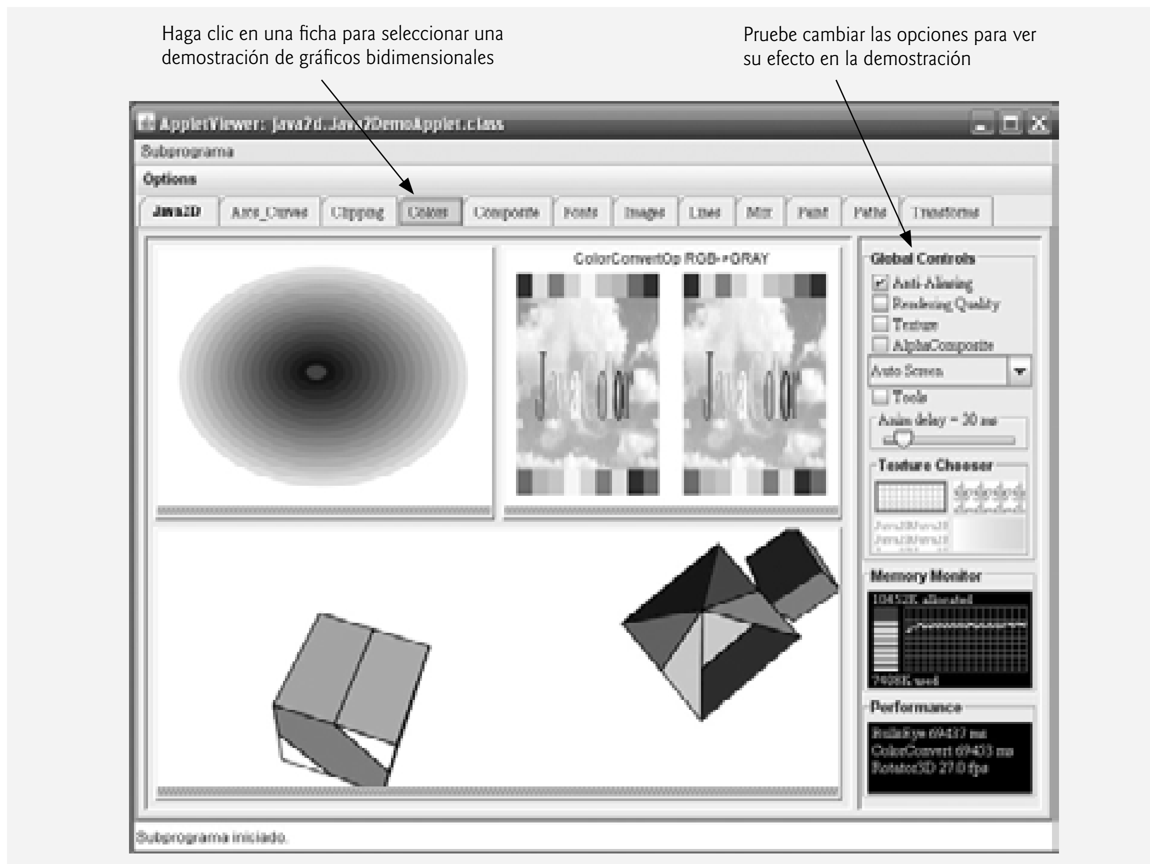


Figura 20.5 | Ejecución de ejemplo del applet `Java2D`.

20.3 Applet simple en Java: cómo dibujar una cadena

Todo applet en Java es una interfaz gráfica de usuario, en la que podemos colocar componentes de GUI mediante el uso de las técnicas presentadas en el capítulo 11, o dibujar mediante el uso de las técnicas demostradas en el

capítulo 12. En este capítulo demostraremos cómo dibujar en un applet. Los ejemplos en los capítulos 21, 23 y 24 demuestran cómo crear la interfaz gráfica de usuario de un applet.

Ahora crearemos nuestro propio applet. Comenzaremos con un applet simple (figura 20.6) que dibuja `"Bienvenido a la programación en Java!"`. En la figura 20.7 se muestra a este applet ejecutándose en dos contenedores de applets: el `appletviewer` y el explorador Web Microsoft Internet Explorer. Al final de esta sección explicaremos cómo ejecutar el applet en un explorador Web.

```java
// Fig. 20.6: BienvenidoApplet.java
// Su primer applet en Java.
import java.awt.Graphics;   // el programa utiliza la clase Graphics
import javax.swing.JApplet; // el programa utiliza la clase JApplet

public class BienvenidoApplet extends JApplet
{
   // dibuja el texto en el fondo del applet
   public void paint( Graphics g )
   {
      // llama a la versión del método paint de la superclase
      super.paint( g );

      // dibuja un objeto String en la coordenada x 25 y la coordenada y 25
      g.drawString( "Bienvenido a la programacion en Java!", 25, 25 );
   } // fin del método paint
} // fin de la clase BienvenidoApplet
```

Figura 20.6 | Applet que dibuja una cadena.

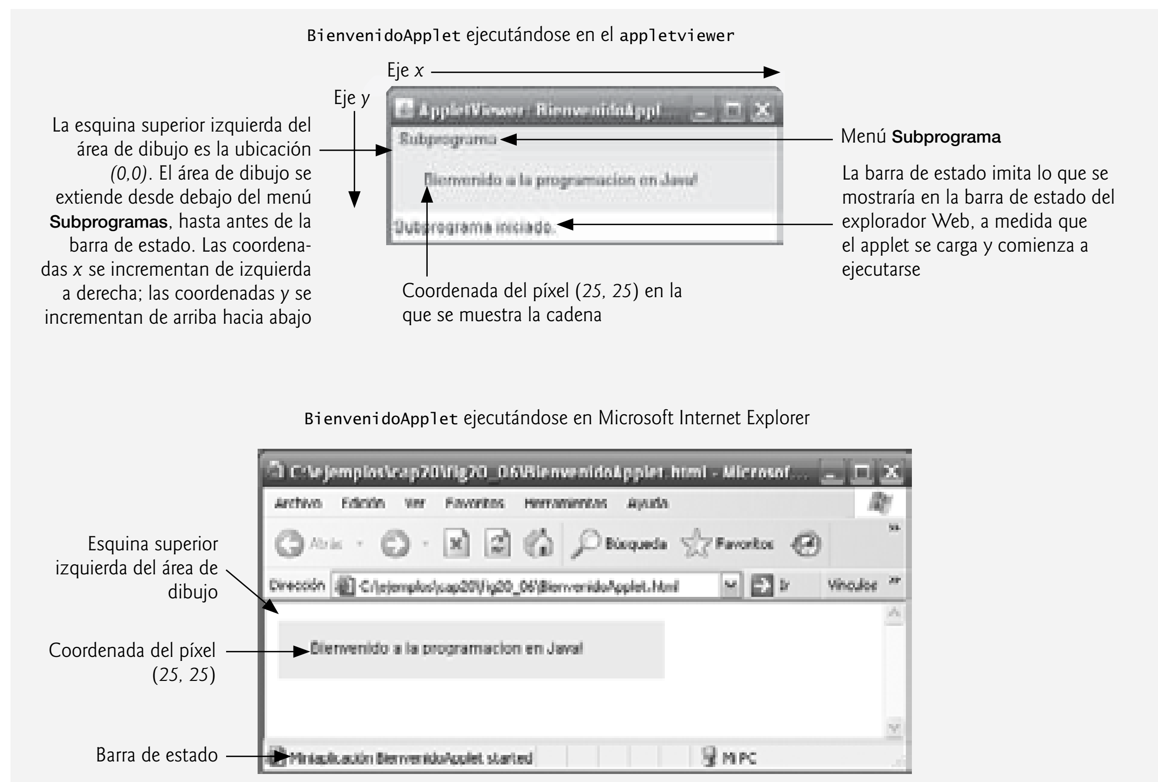


Figura 20.7 | Resultados de ejemplo del applet `BienvenidoApplet` en la figura 20.6.

Creación de la clase Applet

En la línea 3 se importa la clase `Graphics` para permitir al applet dibujar gráficos, como líneas, rectángulos, óvalos y cadenas de caracteres. La clase `JApplet` (que se importa en la línea 4) del paquete `javax.swing` se utiliza para crear applets. Al igual que con las aplicaciones, todo applet de Java contiene por lo menos una declaración de clase `public`. Un contenedor de applets sólo puede crear objetos de clases que sean `public` y extiendan a `JApplet` ¿o a la clase `Applet` de las versiones anteriores de Java? Por esta razón, la clase `BienvenidoApplet` (líneas 6 a 17) extiende a `JApplet`.

Un contenedor de applets espera que todo applet de Java tenga métodos llamados **`init`**, **`start`**, **`paint`**, **`stop`** y **`destroy`**, cada uno de los cuales está declarado en la clase `JApplet`. Cada nueva clase de applet que crea el programador hereda las implementaciones predeterminadas de estos métodos de la clase `JApplet`. Estos métodos se pueden sobrescribir (redefinir) para realizar tareas específicas de cada applet. En la sección 20.4 veremos cada uno de estos métodos con más detalle.

Cuando un contenedor de applets carga la clase `BienvenidoApplet`, el contenedor crea un objeto de tipo `BienvenidoApplet`, y después llama a tres de los métodos del applet. En secuencia, estos tres métodos son: `init`, `start` y `paint`. Si no declaramos estos métodos en el applet, el contenedor de applets llama a las versiones heredadas. Los métodos `init` y `start` de la superclase tienen cuerpos vacíos, por lo que no realizan ninguna tarea. El método `paint` de la superclase no dibuja nada en el applet.

Tal vez usted se pregunte por qué es necesario heredar los métodos `init`, `start` y `paint` si sus implementaciones predeterminadas no realizan tareas. Algunos applets no utilizan todos estos métodos. Sin embargo, el contenedor de applets no sabe eso. Por ende, espera que todo applet tenga estos métodos, de manera que pueda proporcionar una secuencia de inicio consistente para cada applet. Esto es similar al hecho de que las aplicaciones siempre empiezan su ejecución con `main`. Al heredar las versiones "predeterminadas" de estos métodos, se garantiza que el contenedor de applets podrá ejecutar cada applet de manera uniforme. Además, al heredar las implementaciones predeterminadas de estos métodos, el programador puede concentrarse en definir sólo los métodos requeridos para un applet específico.

Cómo sobrescribir el método **`paint`** *para dibujar*

Para permitir que nuestro applet dibuje, la clase `BienvenidoApplet` sobrescribe el método `paint` (líneas 9 a 16), al colocar instrucciones en el cuerpo de `paint` que dibujan un mensaje en la pantalla. El método `paint` recibe un parámetro de tipo `Graphics` (al cual se le llama `g` por convención), el cual se utiliza para dibujar gráficos en el applet. No se llama explícitamente al método `paint` en un applet. En vez de ello, el contenedor de applets llama a `paint` para indicar al applet cuándo dibujar, y el contenedor de applets es responsable de pasar un objeto `Graphics` como argumento.

En la línea 12 se hace una llamada a la versión del método `paint` de la superclase, que se heredó de `JApplet`. Esta instrucción debe ser la primera instrucción en el método `paint` de todo applet. Si se omite podría provocar errores sutiles de dibujo en los applets que combinen el dibujo con componentes de la GUI.

En la línea 15 se utiliza el método `drawString` de `Graphics` para dibujar `Bienvenido a la programacion en Java!` en el applet. Este método recibe como argumentos el objeto `String` a dibujar y las coordenadas *x-y* en las que debe aparecer la esquina inferior izquierda del objeto `String` en el área de dibujo. Cuando se ejecuta la línea 15, dibuja el objeto `String` en el applet, en las coordenadas `25` y `25`.

20.3.1 Cómo ejecutar un applet en el `appletviewer`

Al igual que con las clases de aplicaciones, debemos compilar una clase de applet antes de poder ejecutarla. Después de crear la clase `BienvenidoApplet` y guardarla en el archivo `BienvenidoApplet.java`, abra una ventana de comandos, cambie al directorio en el que guardó la declaración de la clase de applet y compile la clase `BienvenidoApplet`.

Recuerde que los applets están incrustados en páginas Web para ejecutarlos en un contenedor de applets (`appletviewer` o un explorador Web). Antes de poder ejecutar el applet, debe crear un documento HTML (Lenguaje de marcado de hipertexto) que especifique cuál applet ejecutar en el contenedor de applets. Por lo general, un documento HTML termina con la extensión de archivo "**`.html`**" o "**`.htm`**". En la figura 20.8 se muestra un documento HTML simple (`BienvenidoApplet.html`) que carga el applet definido en la figura 20.6, en un contenedor de applets. [*Nota:* si está interesado en aprender más acerca de HTML, el CD que se incluye en este

libro contiene tres capítulos de nuestro libro *Internet and World Wide Web How to Program, Tercera edición*, que introducen la versión actual de HTML (conocida como XHTML) y la herramienta de formato de páginas Web conocida como Hojas de estilo en cascada (CSS)].

La mayoría de los elementos de HTML se delimitan mediante pares de **etiquetas**. Por ejemplo, las líneas 1 y 4 de la figura 20.8 indican el inicio y el fin, respectivamente, del documento HTML. Todas las etiquetas de HTML empiezan con un signo <, y terminan con un signo >. En las líneas 2 y 3 se especifica un **elemento applet**, el cual indica al contenedor de applets que debe cargar un applet específico y define el tamaño del área de visualización del applet (su anchura y altura en píxeles) en el contenedor de applets. Por lo general, el applet y su correspondiente documento HTML se almacenan en el mismo directorio en el disco. Comúnmente, un explorador Web carga un documento HTML de una computadora (distinta a la del lector) conectada a Internet. Sin embargo, los documentos HTML también pueden residir en su computadora (como vio en la sección 20.2). Cuando un contenedor de applets encuentra un documento HTML que contiene un applet, el contenedor carga de manera automática el archivo (o archivos) .class del applet, del mismo directorio en la computadora en donde reside el documento HTML.

El elemento applet tiene varios **atributos**. El primer atributo en la línea 2, code = "BienvenidoApplet.class", indica que el archivo BienvenidoApplet.class contiene la clase de applet compilada. Los atributos segundo y tercero en la línea 2 indican la anchura (**width**) de 300 y la altura (**height**) de 45 del applet, en píxeles. La etiqueta </applet> (línea 3) termina el elemento applet que empezó en la línea 2. La etiqueta </html> (línea 4) termina el documento HTML.



Observación de apariencia visual 20.1

Para asegurar que un applet pueda verse apropiadamente en la mayoría de las pantallas de computadora, generalmente debe ser menor de 1024 píxeles de ancho y de 768 píxeles de alto (las medidas soportadas por la mayoría de las pantallas de computadora).



Error común de programación 20.1

Olvidar la etiqueta </applet> *de cierre evita que el applet se ejecute en ciertos contenedores de applets. El* applet-viewer *termina sin indicar un error. Algunos exploradores Web simplemente ignoran el elemento* applet *incompleto.*

```
1  <html>
2  <applet code = "BienvenidoApplet.class" width = "300" height = "45">
3  </applet>
4  </html>
```

Figura 20.8 | BienvenidoApplet.html carga a BienvenidoApplet (figura 20.6) en un contenedor de applets.



Tip para prevenir errores 20.1

Si recibe un mensaje de error MissingResourceException *al cargar un applet en el* appletviewer *o en un explorador Web, compruebe la etiqueta* <applet> *en el documento HTML cuidadosamente para detectar errores de sintaxis, como las comas (,) entre los atributos.*

El appletviewer sólo comprende las etiquetas de HTML <applet> y </applet> e ignora a todas las demás etiquetas en el documento. El appletviewer es un sitio ideal para probar un applet y asegurar que se ejecute en forma apropiada. Una vez que se verifique la ejecución del applet, puede agregar sus etiquetas de HTML a una página Web que otros puedan ver en sus exploradores Web.

Para ejecutar BienvenidoApplet en el appletviewer, abra una ventana de comandos, cambie al directorio que contiene su applet y su documento HTML, y después escriba

```
appletviewer BienvenidoApplet.html
```



Tip para prevenir errores 20.2

Pruebe sus applets en el contenedor de applets `appletviewer` *antes de ejecutarlos en un explorador Web. A menudo, los exploradores Web guardan una copia de un applet en memoria, hasta que se cierran todas sus ventanas. Si modifica un applet, lo vuelve a compilar y después lo carga en su explorador Web, éste podría seguir ejecutando la versión original del applet. Cierre todas las ventanas del explorador Web para eliminar el applet anterior de la memoria. Abra una nueva ventana del explorador Web y cargue su applet para ver los cambios.*



Tip para prevenir errores 20.3

Pruebe sus applets en todos los exploradores Web en los que se ejecutará, para asegurar que operen en forma correcta.

20.3.2 Ejecución de un applet en un explorador Web

Las ejecuciones de los programas de ejemplo de la figura 20.6 demuestran cómo se ejecuta `BienvenidoApplet` en el `appletviewer` y en el explorador Web Microsoft Internet Explorer. Para ejecutar un applet en Internet Explorer, realice los siguientes pasos:

1. Seleccione **Abrir…** en el menú **Archivo**.
2. En el cuadro de diálogo que se despliegue, haga clic en el botón **Examinar…**.
3. Localice el directorio que contenga el documento de HTML para el applet que desee ejecutar.
4. Seleccione el documento de HTML.
5. Haga clic en el botón **Abrir**.
6. Haga clic en el botón **Aceptar**.

[*Nota:* los pasos para ejecutar applets en otros exploradores Web son similares].

Si su applet se ejecuta en el `appletviewer`, pero no se ejecuta en su explorador Web, tal vez Java no esté instalado y configurado para su explorador Web. En este caso, visite el sitio Web `java.com` y haga clic en el botón **Descargar AHORA** para instalar Java para su explorador Web. En Internet Explorer, si esto no corrige el problema, tal vez necesite configurar manualmente Internet Explorer para que utilice Java. Para ello, haga clic en el menú **Herramientas** y seleccione **Opciones de Internet…**, y después haga clic en la ficha **Opciones avanzadas** en la ventana que aparezca. Localice la opción "**Utilizar JRE v1.6.0 para <miniaplicación> (es necesario reiniciar)**" y asegúrese que esté seleccionada, después haga clic en **Aceptar**. Cierre todas las ventanas de su explorador Web antes de volver a intentar ejecutar otro applet.

20.4 Métodos del ciclo de vida de los applets

Ahora que ha creado un applet, vamos a considerar los cinco métodos de applet a los que llama el contenedor de applets, desde el momento en el que se carga el applet en el explorador Web, hasta el momento en el que éste termina el applet. Estos métodos corresponden a diversos aspectos del ciclo de vida de un applet. En la figura 20.9 se enlistan estos métodos, que las clases de applets heredan de la clase `JApplet`. La tabla especifica el momento en el que se llama a cada método y explica su propósito. A excepción del método `paint`, estos métodos tienen cuerpos vacíos de manera predeterminada. Si desea declarar alguno de estos métodos en sus applets y hacer que el contenedor de applets los llame, debe usar los encabezados de los métodos que se muestran en la figura 20.9. Si modifica los encabezados de los métodos (por ejemplo, cambiar los nombres de los métodos o proporcionar

Método	Momento en que se llama al método y su propósito
`public void init()`	
	El contenedor de applets lo llama una vez, cuando se carga un applet para ejecutarlo. Este método inicializa un applet. Las acciones comunes que se realizan aquí son: inicializar campos, crear componentes de GUI, cargar los sonidos a reproducir, cargas las imágenes a visualizar (vea el capítulo 20, Multimedia: applets y aplicaciones) y crear subprocesos (vea el capítulo 23, Subprocesamiento múltiple).

Figura 20.9 | Métodos del ciclo de vida de `JApplet`, que un contenedor de applets llama durante la ejecución de un applet. (Parte 1 de 2).

Método	Momento en que se llama al método y su propósito
`public void start()`	El contenedor de applets lo llama una vez que el método `init` termina su ejecución. Además, si el usuario explora otro sitio Web y después regresa a la página HTML del applet, el método `start` se llama de nuevo. Este método realiza todas las tareas que deben completarse cuando el applet se carga por primera vez, y que deben realizarse cada vez que se vuelva a visitar la página HTML del applet. Las acciones que se realizan aquí podrían incluir: iniciar una animación (vea el capítulo 21) o iniciar otros subprocesos de ejecución (vea el capítulo 23).
`public void paint( Graphics g )`	El contenedor de applets lo llama después de los métodos `init` y `start`. El método `paint` también se llama cuando el applet necesita volver a visualizarse. Por ejemplo, si el usuario cubre el applet con otra ventana abierta en la pantalla, y más adelante descubre el applet, se hace una llamada al método `paint`. Las acciones comunes que se realizan aquí incluyen: dibujar con el objeto `Graphics g` que el contenedor de applets pasa al método `paint`.
`public void stop()`	El contenedor de applets lo llama cuando el usuario sale de la página Web del applet para ir a explorar otra página Web. Como es posible que el usuario regrese a la página Web que contiene el applet, el método `stop` realiza tareas que podrían requerirse para suspender la ejecución del applet, de manera que no utilice tiempo de procesamiento de la computadora cuando no esté visualizado en la pantalla. Las acciones comunes que se realizan en este método detendrían la ejecución de animaciones y subprocesos.
`public void destroy()`	El contenedor de applets lo llama cuando el applet se va a eliminar de la memoria. Esto ocurre cuando el usuario sale de la sesión de navegación, cerrando todas las ventanas del explorador Web, y también puede ocurrir a discreción del explorador Web, cuando el usuario ha navegado hacia otras páginas Web. El método realiza cualquier tarea que se requiera para limpiar los recursos asignados al applet.

Figura 20.9 | Métodos del ciclo de vida de `JApplet`, que un contenedor de applets llama durante la ejecución de un applet. (Parte 2 de 2).

parámetros adicionales), el contenedor de applets no llamará a sus métodos. En vez de ello, llamará a los métodos de la superclase, heredados de `JApplet`.



Error común de programación 20.2

Si declara los métodos `init`, `start`, `paint`, `stop` o `destroy` con encabezados que sean distintos a los que se muestran en la figura 20.9, el contenedor de applets no los llamará. El código especificado en sus versiones de los métodos no se ejecutará.

20.5 Cómo inicializar una variable de instancia con el método `init`

Nuestro siguiente applet (figura 20.10) calcula la suma de dos valores introducidos por el usuario, y muestra el resultado arrastrando un objeto `String` dentro de un rectángulo en el applet. La suma se almacena en una varia-

```
// Fig. 20.10: SumaApplet.java
// Suma de dos números de punto flotante.
import java.awt.Graphics;       // el programa usa la clase Graphics
import javax.swing.JApplet;     // el programa usa la clase JApplet
import javax.swing.JOptionPane; // el programa usa la clase JOptionPane
```

Figura 20.10 | Suma de valores `double`. (Parte 1 de 2).

```java

public class SumaApplet extends JApplet
{
   private double suma; // suma de los valores introducidos por el usuario

   // inicializa el applet, obteniendo los valores del usuario
   public void init()
   {
      String primerNumero;  // primera cadena introducida por el usuario
      String segundoNumero; // segunda cadena introducida por el usuario

      double numero1; // primer número a sumar
      double numero2; // segundo número a sumar

      // obtiene el primer número del usuario
      primerNumero = JOptionPane.showInputDialog(
         "Escriba el primer valor de punto flotante" );

      // obtiene el segundo número del usuario
      segundoNumero = JOptionPane.showInputDialog(
         "Escriba el segundo valor de punto flotante" );

      // convierte los números del tipo String al tipo double
      numero1 = Double.parseDouble( primerNumero );
      numero2 = Double.parseDouble( segundoNumero );

      suma = numero1 + numero2; // suma los números
   } // fin del método init

   // dibuja los resultados en un rectángulo, en el fondo del applet
   public void paint( Graphics g )
   {
      super.paint( g ); // llama a la versión del método paint de la superclase

      // dibuja un rectángulo empezando desde (15, 10), que tenga 270
      // píxeles de ancho y 20 píxeles de alto
      g.drawRect( 15, 10, 270, 20 );

      // dibuja los resultados como un objeto String en (25, 25)
      g.drawString( "La suma es " + suma, 25, 25 );
   } // fin del método paint
} // fin de la clase SumaApplet
```

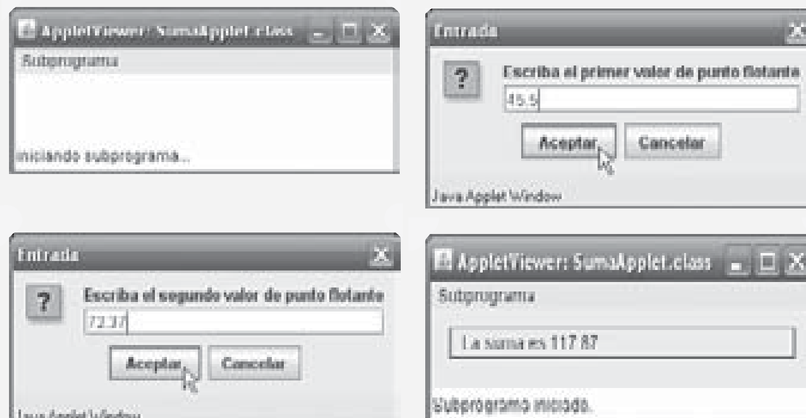


Figura 20.10 | Suma de valores `double`. (Parte 2 de 2).

ble de instancia de la clase SumaApplet, para que pueda utilizarse tanto en el método init como en el método paint. El documento HTML que carga este applet en el appletviewer se muestra en la figura 20.11.

```
<html>
<applet code = "SumaApplet.class" width = "300" height = "50">
</applet>
</html>
```

Figura 20.11 | SumaApplet.html carga la clase SumaApplet de la figura 20.10 dentro de un contenedor de applets.

El applet solicita al usuario dos números de punto flotante. En la línea 9 (figura 20.10) se declara la variable de instancia suma de tipo double. El applet contiene dos métodos: init (líneas 12 a 33) y paint (líneas 36 a 46). Cuando un contenedor de applets carga este applet, el contenedor crea una instancia de la clase SumaApplet y llama a su método init; esto ocurre sólo una vez durante la ejecución de un applet. Por lo común, el método init inicializa los campos del applet (si necesitan inicializarse con valores que no sean los predeterminados) y realiza otras tareas que sólo deben ocurrir una vez, cuando el applet empieza a ejecutarse. La primera línea de init siempre aparece como se muestra en la línea 12, la cual indica que init es un método public que no recibe argumentos y no devuelve información cuando termina su ejecución.

En las líneas 14 a 30 se declaran variables para almacenar los valores introducidos por el usuario, obtener la entrada del usuario y convertir los objetos String introducidos por el usuario en valores double.

La instrucción de asignación en la línea 32 suma los valores almacenados en las variables numero1 y numero2, y asigna el resultado a la variable de instancia suma. En este punto, el método init del applet devuelve el control del programa al contenedor de applets, el cual a su vez llama al método start del applet. No declaramos a start en este applet, por lo que aquí se hace la llamada al método heredado de la clase JApplet. En los capítulos 21 y 23 veremos usos comunes del método start.

A continuación, el contenedor de applets llama al método paint del applet, el cual dibuja un rectángulo (línea 42) en donde aparecerá el resultado de la suma. En la línea 45 se hace una llamada al método drawString del objeto Graphics para visualizar los resultados. La instrucción concatena el valor de la variable de instancia suma al objeto String "La suma es " y muestra en pantalla el objeto String concatenado.



Observación de ingeniería de software 20.1

Las únicas instrucciones que se deben colocar en el método init *de un applet son las que deben ejecutarse sólo una vez, al inicializar el applet.*

20.6 Modelo de seguridad "caja de arena"

Sería peligroso permitir a los applets que, por lo general, se descargan de Internet, leer y escribir archivos en una computadora cliente, o acceder a otros recursos del sistema. Por ejemplo, ¿qué ocurriría si usted descargara un applet malicioso? La plataforma Java utiliza el **modelo de seguridad "caja de arena"** para evitar que el código que usted descargue en su equipo local acceda a los recursos del sistema local, como los archivos. El código que se ejecuta dentro de la "caja de arena" no tiene permitido "jugar fuera de la caja". Para obtener más información acerca de la seguridad y los applets, visite

developer.java.sun.com/developer/technicalArticles/Security/Signed

Para obtener información acerca del modelo de seguridad de la Plataforma Java 2, visite

java.sun.com/javase/6/docs/technotes/guides/security/index.html

20.7 Recursos en Internet y Web

Si tiene acceso a Internet, hay una gran cantidad de recursos sobre applets de Java disponibles para usted. El mejor lugar para empezar es el de origen: el sitio Web de Java de Sun Microsystems, java.sun.com. La página Web

java.sun.com/applets

contiene varios recursos sobre applets de Java, incluyendo los applets de muestra del JDK y otros applets (muchos de los cuales se pueden descargar).

Si no tiene Java instalado y configurado para su explorador Web, puede visitar

`java.com`

y hacer clic en el botón **Descargar AHORA** para descargar e instalar Java en su explorador Web. Se proporcionan instrucciones para varias versiones de Windows, Linux, Solaris y Mac OS.

El sitio Web de Java de Sun Microsystems

`java.sun.com`

incluye soporte técnico, foros de discusión, artículos técnicos, recursos, anuncios de nuevas características de Java y un acceso anticipado a las nuevas tecnologías de Java.

Para ver varios tutoriales en línea gratuitos, visite el sitio Web

`java.sun.com/learning`

Otro sitio Web útil es **JARS** (conocido originalmente como **Servicio de clasificación de applets de Java**). El sitio Web de JARS

`www.jars.com`

era un almacén de applets de Java que clasificaba cada applet registrado en el sitio, de manera que los visitantes pudieran ver los mejores applets en Web. En las primeras etapas del desarrollo del lenguaje Java, lograr que su applet se clasificara aquí era una excelente forma de demostrar sus habilidades de programación en Java. JARS es ahora un sitio de clasificaciones sobre todo lo relacionado con Java para programadores.

Los recursos que se enlistan en esta sección proporcionan hipervínculos hacia muchos otros sitios Web relacionados con Java. Invierta tiempo explorando estos sitios, ejecutando applets y leyendo el código fuente de éstos cuando esté disponible. Esto le ayudará a expandir con rapidez su conocimiento sobre Java.

20.8 Conclusión

En este capítulo aprendió los fundamentos de los applets de Java. Aprendió los conceptos básicos sobre HTML, que le permiten incrustar un applet en una página Web y ejecutarlo en un contenedor de applets, como `appletviewer` o un explorador Web. Además, aprendió acerca de los cinco métodos que el contenedor de applets llama de manera automática durante el ciclo de vida de un applet. En el siguiente capítulo verá varios applets adicionales, a medida que vayamos presentando las herramientas básicas de multimedia. En el capítulo 23, Subprocesamiento múltiple, verá un applet con los métodos `start` y `stop` que se utilizan para controlar varios subprocesos de ejecución. En el capítulo 24, Redes, le demostraremos cómo personalizar un applet a través de parámetros que se especifiquen en un elemento HTML del `applet`.

Resumen

Sección 20.1 Introducción

- Los applets son programas en Java que pueden incrustarse en documentos HTML.
- Cuando un explorador Web carga una página Web que contiene un applet, éste se descarga en el explorador Web y se ejecuta.
- El explorador Web que ejecuta un applet se conoce como el contenedor de applets. El JDK incluye el contenedor de applets `appletviewer`, para probar applets antes de incrustarlos en una página Web.

Sección 20.2 Applets de muestra incluidos en el JDK

- Para volver a ejecutar un applet en el `appletviewer`, haga clic en el menú **Subprograma** y seleccione el elemento de menú **Volver a cargar**.
- Para terminar el `appletviewer`, seleccione el elemento de menú **Salir** del menú **Subprograma**.

Sección 20.3 Applet simple en Java: cómo dibujar una cadena

- Todo applet de Java es una interfaz gráfica de usuario, en la cual podemos colocar componentes de GUI o dibujar.
- La clase `JApplet` del paquete `javax.swing` se utiliza para crear applets.
- Un contenedor de applets sólo puede crear objetos de clases que sean `public` y extiendan a `JApplet` (o la clase `Applet` de las versiones anteriores de Java).
- Un contenedor de applets espera que cada applet de Java tenga métodos llamados `init`, `start`, `paint`, `stop` y `destroy`, cada uno de los cuales está declarado en la clase `JApplet`. Cada nueva clase de applet que usted cree hereda las implementaciones predeterminadas de estos métodos de la clase `JApplet`.
- Cuando un contenedor de applets carga un applet, el contenedor crea un objeto del tipo del applet, y después llama a los métodos `init`, `start` y `paint` del applet. Si no declara estos métodos en su applet, el contenedor de applets llama a las versiones heredadas.
- Los métodos `init` y `start` de la superclase tienen cuerpos vacíos, por lo cual no realizan ninguna tarea. El método `paint` de la superclase no dibuja nada en el applet.
- Para permitir a un applet dibujar, hay que sobrescribir su método `paint`. No debemos llamar explícitamente al método `paint` en un applet. En vez de ello, el contenedor de applet debe llamar a `paint` para indicar al applet cuándo debe dibujar, y el contenedor de applets es responsable de pasarle un objeto `Graphics` como argumento.
- La primera instrucción en el método `paint` debe ser una llamada al método `paint` de la superclase. Omitir esto puede provocar errores sutiles de dibujo en applets que combinan el dibujo y componentes de GUI.
- Antes de ejecutar un applet, debemos crear un documento HTML (Lenguaje de marcado de hipertexto) que especifique cuál applet ejecutar en el contenedor de applets. Por lo general, un documento HTML termina con una extensión de archivo "`.html`" o "`.htm`".
- La mayoría de los elementos de HTML se delimitan mediante pares de etiquetas. Todas las etiquetas de HTML empiezan con un signo `<`, y terminan con un signo `>`.
- Un elemento `applet` indica al contenedor de applets que cargue un applet específico, y define el tamaño del área de visualización del applet (su anchura y altura en píxeles) en el contenedor de applets.
- Por lo general, un applet y su correspondiente documento HTML se guardan en el mismo directorio.
- Comúnmente, un explorador Web carga un documento HTML de una computadora (distinta a la del lector) conectada a Internet.
- Cuando un contenedor de applets encuentra un documento HTML que contiene un applet, carga de manera automática el (los) archivo(s) `.class` del applet desde el mismo directorio en la computadora en la que reside el documento HTML.
- El `appletviewer` sólo comprende las etiquetas `<applet>` y `</applet>` de HTML, e ignora a todas las demás etiquetas en el documento.
- El `appletviewer` es un sitio ideal para probar un applet y asegurarse de que se ejecute apropiadamente. Una vez que se verifica la ejecución del applet, se pueden agregar sus etiquetas HTML a una página Web que otros puedan ver en sus exploradores Web.

Sección 20.4 Métodos del ciclo de vida de los applets

- Hay cinco métodos de applet que el contenedor de applets llama, desde el momento en el que se carga el applet en el explorador Web, hasta el momento en que el explorador Web termina el applet. Estos métodos corresponden a varios aspectos del ciclo de vida de un applet.
- El contenedor de applets llama una vez al método `init`, cuando se carga un applet para ejecutarlo. Este método inicializa el applet.
- El contenedor de applets llama al método `start` una vez que el método `init` termina de ejecutarse. Además, si el usuario navega hacia otro sitio Web y después regresa a la página HTML del applet, se llama otra vez al método `start`.
- El contenedor de applets llama al método `paint` después de los métodos `init` y `start`. El método `paint` también se llama cuando el applet necesita volver a dibujarse.
- El contenedor de applets llama al método `stop` cuando el usuario sale de la página Web del applet, al navegar hacia otra página Web.
- El contenedor de applets llama al método `destroy` cuando el applet se va a eliminar de la memoria. Esto ocurre cuando el usuario sale de la sesión de navegación, al cerrar todas las ventanas del explorador Web, y también puede ocurrir a discreción del explorador Web, cuando el usuario navega hacia otras páginas Web.

Terminología

.htm, extensión de archivo
.html, extensión de archivo
<applet>, etiqueta
altura (height) de un applet
anchura (width) de un applet
applet
appletviewer
atributo
contenedor de applets
demo, directorio del JDK
elemento de HTML
elemento de HTML de un applet
etiqueta
init, método de JApplet
JApplet, clase
Lenguaje de marcado de hipertexto (HTML)
paint, método de JApplet
parseDouble, método de Double
Salir, elemento de menú en el appletviewer
signo <
signo >
start, método de JApplet
stop, método de JApplet
Subprograma, menú en el appletviewer
Volver a cargar, elemento de menú en el appletviewer

Ejercicio de autoevaluación

20.1 Complete las siguientes oraciones:

a) Los applets de Java empiezan a ejecutarse con una serie de llamadas a tres métodos: __________________, __________________ y __________________.
b) El método __________________ se invoca para un applet cada vez que el usuario de un explorador Web sale de la página HTML en la que reside el applet.
c) Todo applet debe extender a la clase __________________.
d) El __________________ o un explorador Web pueden utilizarse para ejecutar un applet de Java.
e) El método __________________ se llama cada vez que el usuario de un explorador Web vuelve a visitar la página HTML en la que reside un applet.
f) Para cargar un applet en un explorador Web, debe primero definir un archivo __________________.
g) El método __________________ se llama una vez cuando el applet empieza a ejecutarse.
h) El método __________________ se invoca para dibujar en un applet.
i) El método __________________ se invoca para un applet cuando el explorador Web lo elimina de la memoria.
j) Las etiquetas __________________ y __________________ de HTML especifican que debe cargarse un applet en un contenedor de applets, y ejecutarse.

Respuestas a los ejercicios de autoevaluación

20.1 a) init, start, paint. b) stop. c) JApplet (o Applet). d) appletviewer. e) start. f) HTML. g) init. h) paint. i) destroy. j) <applet>, </applet>.

Ejercicios

20.2 Escriba un applet que pida al usuario que introduzca dos números de punto flotante, que obtenga los dos números del usuario y dibuje su suma, producto (multiplicación), diferencia y cociente (división). Use las técnicas que se muestran en la figura 20.10.

20.3 Escriba un applet que pida al usuario que introduzca dos números de punto flotante, que obtenga los números del usuario y muestre los dos números primero, y después el número más grande seguido de las palabras "es mayor que" como una cadena en el applet. Si los números son iguales, el applet deberá imprimir el mensaje "Estos numeros son iguales". Use las técnicas que se muestran en la figura 20.10.

20.4 Escriba un applet que reciba tres números de punto flotante del usuario y que muestre la suma, el promedio, el producto, el menor y el mayor de estos números, como cadenas en el applet. Use las técnicas que se muestran en la figura 20.10.

20.5 Escriba un applet que pida al usuario que introduzca el radio de un círculo como un número de punto flotante, y que dibuje el diámetro, circunferencia y área del círculo. Use el valor 3.14159 para π. Use las técnicas que se muestran en la figura 20.10. [*Nota:* también puede usar la constante predefinida Math.PI para el valor de π. Esta constante es

más precisa que el valor 3.14159. La clase `Math` se define en el paquete `java.lang`, por lo que no necesita importarla]. Use las siguientes fórmulas (*r* es el radio):

diámetro = $2r$
circunferencia = $2\pi r$
área = πr^2

20.6 Escriba un applet que lea cinco enteros, determine cuáles son el mayor y el menor en el grupo, y que los imprima. Use sólo las técnicas de programación que aprendió en este capítulo y en el capítulo 2. Dibuje los resultados en el applet.

20.7 Escriba un applet que dibuje un patrón de tablero de damas, como se muestra a continuación:

```
* * * * * * * *
 * * * * * * * *
* * * * * * * *
 * * * * * * * *
* * * * * * * *
 * * * * * * * *
* * * * * * * *
 * * * * * * * *
```

20.8 Escriba un applet que dibuje rectángulos de distintos tamaños y posiciones.

20.9 Escriba un applet que permita al usuario introducir valores para los argumentos requeridos por el método `drawRect`, y que después dibuje un rectángulo usando los cuatro valores de entrada.

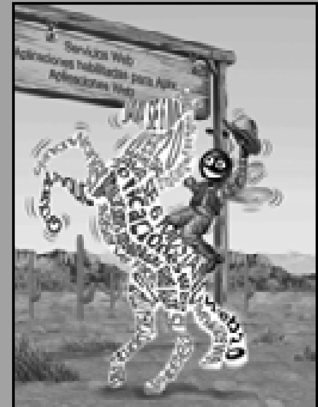
20.10 La clase `Graphics` contiene el método `drawOval`, el cual recibe como argumentos los mismos cuatro argumentos que el método `drawRect`. Los argumentos para el método `drawOval` especifican el "cuadro delimitador" para el óvalo; los lados del cuadro delimitador son los límites del óvalo. Escriba un applet en Java que dibuje un óvalo y un rectángulo con los mismos cuatro argumentos. El óvalo debe tocar el rectángulo en el centro de cada lado.

20.11 Modifique la solución al ejercicio 20.10 para imprimir óvalos de distintas formas y tamaños.

20.12 Escriba un applet que permita al usuario introducir los cuatro argumentos requeridos por el método `drawOval`, y que después dibuje un óvalo usando los cuatro valores de entrada.

21

Multimedia: applets y aplicaciones



La rueda que rechina más fuerte... es la que requiere la grasa.

—John Billings (Henry Wheeler Shaw)

Utilizaremos una señal que he probado que tiene gran alcance y es fácil de gritar. ¡Yajuuuu!

—Zane Grey

Hay un movimiento de vaivén natural en un pez dorado.

—Walt Disney

Entre el movimiento y el acto cae la sombra.

—Thomas Stearns Eliot

OBJETIVOS

En este capítulo aprenderá a:

- Obtener, mostrar y escalar imágenes.
- Crear animaciones a partir de secuencias de imágenes.
- Crear mapas de imágenes.
- Obtener sonidos, reproducirlos, hacer que se reproduzcan indefinidamente y detenerlos mediante el uso de un objeto `AudioClip`.
- Reproducir video mediante la interfaz `Player`.

Plan general

21.1 Introducción

Bienvenido a lo que podría ser la mayor revolución en la historia de la industria computacional. Todos los que entramos al campo hace décadas, estábamos interesados en usar las computadoras principalmente para realizar cálculos aritméticos a grandes velocidades. A medida que el campo de la computación fue evolucionando, empezamos a darnos cuenta de que las herramientas de manipulación de datos de las computadoras son igualmente importantes. El "atractivo" de Java es **multimedia** (el uso de **sonido**, **imágenes**, **gráficos** y **video** para hacer que las aplicaciones "cobren vida"). Aunque la mayor parte del contenido multimedia en Java es bidimensional, los programadores de Java ya pueden utilizar la **API Java 3D** para crear aplicaciones importantes con gráficos en 3D (Sun ofrece un tutorial en línea para la API Java 3D en `java.sun.com/developer/onlineTraining/java3d).`

La programación de multimedia ofrece muchos nuevos retos. El campo es de por sí enorme, y crece rápidamente. La mayoría de las nuevas computadoras que se venden actualmente están "listas para multimedia", con unidades de CD-RW o DVD, tarjetas de audio e incluso con herramientas especiales de video. Las computadoras de escritorio y portátiles económicas son tan poderosas que pueden almacenar y reproducir video y sonido con calidad de DVD, y esperamos ver aún más avances en los tipos de herramientas de multimedia programables, disponibles a través de los lenguajes de programación. Algo que hemos aprendido es a planear para "lo imposible"; en los campos de la computación y las comunicaciones, lo "imposible" se ha vuelto realidad en repetidas ocasiones.

Entre los usuarios que desean gráficos, muchos ahora desean gráficos de color tridimensionales, de alta resolución. El despliegue de imágenes tridimensionales verdaderas tal vez esté disponible dentro de la siguiente década. Imagine tener una televisión tridimensional, de alta resolución tipo cine. ¡Los eventos deportivos y de entretenimiento parecerán estar llevándose a cabo en la sala de su casa! Los estudiantes de medicina en todo el mundo verán cómo se realizan operaciones a miles de millas de distancia, como si estuvieran ocurriendo en la misma habitación. Las personas aprenderán a conducir con simuladores de conducción extremadamente realistas en sus hogares antes de ponerse al volante. Las posibilidades son emocionantes e interminables.

La multimedia demanda un poder computacional extraordinario. Hasta hace poco, no había computadoras a precios accesibles con ese tipo de poder. Los procesadores ultrarrápidos de la actualidad hacen posible la multimedia con efectividad. Las industrias de la computación y la comunicación serán los principales beneficiarios de la revolución multimedia. Los usuarios estarán dispuestos a pagar por procesadores más veloces, memorias más grandes y anchos de banda de comunicación más amplios, que soporten las exigencias de las aplicaciones multimedia. Irónicamente, los usuarios tal vez no tengan que pagar más, ya que la feroz competencia en estas industrias hace que los precios bajen.

Necesitamos lenguajes de programación que faciliten la creación de aplicaciones multimedia. La mayoría de los lenguajes de programación no tienen herramientas multimedia integradas . Sin embargo, a través de sus bibliotecas de clases, Java proporciona extensas características multimedia que le permitirán empezar a desarrollar poderosas aplicaciones multimedia inmediatamente.

En este capítulo presentamos varios ejemplos de las interesantes características multimedia que usted necesitará para crear útiles aplicaciones, incluyendo las siguientes:

1. Los fundamentos de la manipulación de imágenes.

2. Creación de animaciones refinadas.
3. Reproducción de archivos de audio mediante la interfaz `AudioClip`.
4. Creación de mapas de imágenes que pueden detectar cuando el cursor se encuentra sobre ellos, incluso sin hacer clic con el ratón.
5. Reproducción de archivos de video mediante el uso de la interfaz `Player`.

Los ejercicios para este capítulo sugieren docenas de proyectos retadores e interesantes. Cuando creamos estos ejercicios, las ideas no dejaban de surgir. La multimedia impulsa la creatividad en formas que no hemos experimentado con las herramientas "convencionales" de computación. [*Nota:* las herramientas de multimedia de Java van más allá de las que se presentan en este capítulo. Entre estas herramientas se incluyen la **API del marco de trabajo de medios de Java** (**JMF, Java Media Framework API**) para agregar medios de audio y video a una aplicación, la **API de sonido de Java** (**Java Sound API**) para reproducir, grabar y modificar audio; la API 3D de Java (Java 3D API) para crear y modificar gráficos en 3D; la **API de procesamiento avanzado de imágenes de Java** (**Java Advanced Imaging API**) para las herramientas de procesamiento de imágenes, como recortar y escalar; la **API de síntesis de voz de Java** (**Java Speech API**) para introducir comandos del usuario, o emitir comandos de voz al usuario; la API 2D de Java (Java 2D API) para crear y modificar gráficos en 2D, la cual cubrimos en el capítulo 12; y la **API de E/S de imágenes de Java** (**Java Image I/O API**) para leer y escribir imágenes en archivos. En la sección 21.8 se proporcionan vínculos Web para cada una de estas APIs].

21.2 Cómo cargar, mostrar y escalar imágenes

Las herramientas multimedia de Java incluyen gráficos, imágenes, animaciones, sonidos y video. Empezaremos nuestra discusión con las imágenes, y emplearemos distintas en este capítulo. Los desarrolladores pueden crear dichas imágenes con cualquier software para manipulación de imágenes, como Adobe® Photoshop™, Jasc® Paint Shop Pro™ o Microsoft® Paint.

El applet de la figura 21.1 demuestra cómo cargar un objeto **`Image`** (paquete `java.awt`) y cómo cargar un objeto **`ImageIcon`** (paquete `javax.swing`). Ambas clases se utilizan para cargar y mostrar imágenes en pantalla. El applet muestra al objeto **`Image`** en su tamaño original y escalado a un tamaño mayor, utilizando dos versiones del método **`drawImage`** de `Graphics`. También dibuja el objeto `ImageIcon`, utilizando el método **`paintIcon`** del icono. La clase `ImageIcon` implementa a la interfaz `Serializable`, la cual permite escribir fácilmente obje-

```
// Fig. 21.1: CargarImagenYEscalar.java
// Carga una imagen y la muestra en su tamaño original y al doble de su
// tamaño original. Carga y muestra la misma imagen como un objeto ImageIcon.
import java.awt.Graphics;
import java.awt.Image;
import javax.swing.ImageIcon;
import javax.swing.JApplet;

public class CargarImagenYEscalar extends JApplet
{
   private Image imagen1; // crea un objeto Image
   private ImageIcon imagen2; // crea un objeto ImageIcon

   // carga la imagen cuando se carga el applet
   public void init()
   {
      imagen1 = getImage( getDocumentBase(), "floresrojas.png" );
      imagen2 = new ImageIcon( "floresamarillas.png" );
   } // fin del método init

   // muestra la imagen
   public void paint( Graphics g )
```

Figura 21.1 | Cómo cargar y mostrar una imagen en un applet. (Parte 1 de 2).

```
23      {
24         super.paint( g );
25
26         g.drawImage( imagen1, 0, 0, this ); // dibuja la imagen original
27
28         // dibuja la imagen para que se ajuste a la anchura y altura menos 120 píxeles
29         g.drawImage( imagen1, 0, 120, getWidth(), getHeight() - 120, this );
30
31         // dibuja un icono, usando su método paintIcon
32         imagen2.paintIcon( this, g, 180, 0 );
33      } // fin del método paint
34   } // fin de la clase CargarImagenYEscalar
```



Figura 21.1 | Cómo cargar y mostrar una imagen en un applet. (Parte 2 de 2).

tos `ImageIcon` en un archivo, o enviarlos a través de Internet. La clase `ImageIcon` es también más fácil de usar que `Image`, ya que su constructor puede recibir argumentos de varios formatos distintos, incluyendo un arreglo `byte` que contiene los bytes de una imagen, un objeto `Image` ya cargado en memoria, y un objeto `String` o `URL`, los cuales pueden usarse para representar la ubicación de la imagen. Un objeto **`URL`** representa a un Localizador uniforme de recursos, el cual sirve como apuntador a un recurso en World Wide Web, en su computadora o en cualquier equipo conectado en red. Un objeto `URL` se utiliza con más frecuencia cuando se accede a los datos en el equipo actual. Al utilizar un objeto `URL`, el programador también puede acceder a la información en los sitios Web, como cuando se busca información en una base de datos o a través de un motor de búsqueda.

En las líneas 11 y 12 se declaran variables `Image` e `ImageIcon`, respectivamente. La clase `Image` es una clase `abstract`; por lo tanto, el applet no puede crear un objeto de la clase `Image` directamente. En vez de ello, debe llamar a un método que haga que el contenedor de applets cargue y devuelva el objeto `Image` para usarlo en el programa. La clase `Applet` (la superclase directa de `JApplet`) proporciona el método **`getImage`** (línea 17 en el método `init`) para cargar un objeto `Image` en un applet. Esta versión de `getImage` recibe dos argumentos: la ubicación del archivo de la imagen y el nombre de ese archivo. En el primer argumento, el método **`getDocumentBase`** de `Applet` devuelve un objeto `URL` que representa la ubicación de la imagen en Internet (o en su computadora, si el applet se cargó desde ahí). El método **`getDocumentBase`** devuelve la ubicación del archivo HTML como un objeto de la clase `URL`. En conjunto, los dos argumentos especifican el nombre único y la ruta del archivo que se va a cargar (en este caso, el archivo `floresrojas.png` almacenado en el mismo directorio que el archivo HTML que invocó al applet). El segundo argumento especifica el nombre de un archivo de imagen. Java

soporta varios formatos de imágenes, incluyendo **GIF (Formato de Intercambio de Gráficos)**, **JPEG (Grupo Unido de Expertos en Fotografía)** y **PNG (Gráficos Portables de Red)**. Los nombres de archivo para cada uno de estos tipos terminan con `.gif`, `.jpg` (o `.jpeg`) y `.png`, respectivamente.



Tip de portabilidad 21.1

La clase `Image` es una clase `abstract`; como resultado, los programas no pueden instanciar la clase `Image` para crear objetos. Para lograr una independencia de plataformas, la implementación de Java en cada plataforma proporciona su propia subclase de `Image` para almacenar información sobre las imágenes. Los métodos que devuelven referencias a objetos `Image` en realidad devuelven referencias a objetos de la subclase `Image` de la implementación de Java.

En la línea 17 comienza a cargarse la imagen del equipo local (o comienza a descargarse la imagen de Internet). Cuando la imagen es requerida por el programa, se carga en un subproceso de ejecución separado. Recuerde que un subproceso es una actividad en paralelo, y que hablaremos sobre los subprocesos con detalle en el capítulo 23, Subprocesamiento múltiple. Al utilizar un subproceso separado para cargar una imagen, el programa puede continuar su ejecución mientras la imagen se carga. [*Nota:* si el archivo solicitado no está disponible, el método `getImage` no lanza una excepción. Se devuelve un objeto `Image`, pero cuando éste se muestra en pantalla mediante el método `drawImage`, no aparece nada].

La clase `ImageIcon` no es una clase `abstract`; por lo tanto, un programa puede crear un objeto `ImageIcon`. La línea 18 en el método `init` crea un objeto `ImageIcon` que carga la imagen `floresamarillas.png`. La clase `ImageIcon` proporciona varios constructores que permiten a los programas inicializar objetos `ImageIcon` con imágenes del equipo local, o con imágenes almacenadas en Internet.

El método `paint` del applet (líneas 22 a 33) muestra las imágenes. En la línea 26 se utiliza el método `drawImage` de `Graphics` para mostrar un objeto `Image`. El método `drawImage` recibe cuatro argumentos. El primer argumento es una referencia al objeto `Image` que se va a mostrar (`imagen1`). Los argumentos segundo y tercero son las coordenadas *x* y *y* en las que se mostrará la imagen en el applet; las coordenadas indican la ubicación de la esquina superior izquierda de la imagen. El último argumento es una referencia a un objeto **`ImageObserver`**; una interfaz implementada por la clase `Component`. Como la clase `JApplet` extiende a `Component` de manera indirecta, todos los objetos `JApplet` son objetos `ImageObserver`. Este argumento es importante cuando se muestran imágenes extensas que requieren de mucho tiempo para descargarse desde Internet. Es posible que un programa intente mostrar la imagen antes de que ésta se descargue completamente. El objeto `ImageObserver` es notificado para actualizar la imagen que se mostrará, a medida que el objeto `Image` se carga y actualiza la imagen en la pantalla, si ésta no estaba completa cuando se empezó a mostrar. Al ejecutar este applet, observe cuidadosamente cómo se van mostrando piezas de la imagen mientras ésta se carga. [*Nota:* en equipos rápidos, tal vez no pueda percatarse de este efecto].

En la línea 29 se utiliza una versión sobrecargada del método `drawImage` para mostrar una versión *escalada* de la imagen. Los argumentos cuarto y quinto especifican la *anchura* y *altura* de la imagen para mostrarla en pantalla. El método `drawImage` escala la imagen para que se ajuste a la anchura y altura especificadas. En este ejemplo, el cuarto argumento indica que la anchura de la imagen escalada debe ser igual a la anchura del applet, y el quinto argumento indica que la altura debe ser 120 píxeles menor que la altura del applet. La anchura y la altura del applet se determinan mediante llamadas a los métodos `getWidth` y `getHeight` (heredados de la clase `Component`).

En la línea 32 se utiliza el método `paintIcon` de `ImageIcon` para mostrar la imagen. Este método requiere cuatro argumentos: una referencia al objeto `Component` en el que se mostrará la imagen, una referencia al objeto `Graphics` que desplegará la imagen, la coordenada *x* de la esquina superior izquierda de la imagen y la coordenada *y* de la esquina superior izquierda de la imagen.

21.3 Animación de una serie de imágenes

El siguiente ejemplo demuestra cómo animar una serie de imágenes almacenadas en un arreglo de objetos `ImageIcon`. La animación presentada en las figuras 21.2 y 21.3 se implementa mediante el uso de una subclase de `JPanel` llamada `AnimadorLogoJPanel` (figura 21.2) que puede adjuntarse a una ventana de aplicación o a un objeto `JApplet`. La clase `AnimadorLogo` (figura 21.3) también declara un método `main` (líneas 8 a 20 de la figura 21.3) para ejecutar la animación como una aplicación. El método `main` declara una instancia de la clase `JFrame` y adjunta un objeto `AnimadorLogoJPanel` al objeto `JFrame` para mostrar la animación.

```
// Fig. 21.2: AnimadorLogoJPanel.java
// Animación de una serie de imágenes.
import java.awt.Dimension;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.Graphics;
import javax.swing.ImageIcon;
import javax.swing.JPanel;
import javax.swing.Timer;

public class AnimadorLogoJPanel extends JPanel
{
   private final static String NOMBRE_IMAGEN = "deitel"; // nombre de la imagen base
   protected ImageIcon imagenes[]; // arreglo de imágenes
   private final int TOTAL_IMAGENES = 30; // número de imágenes
   private int imagenActual = 0; // índice de la imagen actual
   private final int RETRASO_ANIMACION = 50; // retraso en milisegundos
   private int anchura; // anchura de la imagen
   private int altura; // altura de la imagen

   private Timer temporizadorAnimacion; // objeto Timer que controla la animación

   // el constructor inicializa el objeto AnimadorLogoJPanel, cargando las imágenes
   public AnimadorLogoJPanel()
   {
      imagenes = new ImageIcon[ TOTAL_IMAGENES ];

      // carga 30 imágenes
      for ( int cuenta = 0; cuenta < imagenes.length; cuenta++ )
         imagenes[ cuenta ] = new ImageIcon( getClass().getResource(
            "imagenes/" + NOMBRE_IMAGEN + cuenta + ".gif" ) );

      // este ejemplo supone que todas las imágenes tienen la misma anchura y altura
      anchura = imagenes[ 0 ].getIconWidth(); // obtiene la anchura del icono
      altura = imagenes[ 0 ].getIconHeight(); // obtiene la altura del icono
   } // fin del constructor de AnimadorLogoJPanel

   // muestra la imagen actual
   public void paintComponent( Graphics g )
   {
      super.paintComponent( g ); // llama al método paintComponent de la superclase

      imagenes[ imagenActual ].paintIcon( this, g, 0, 0 );

      // establece la siguiente imagen a dibujar, sólo si el temporizador está funcionando
      if ( temporizadorAnimacion.isRunning() )
         imagenActual = ( imagenActual + 1 ) % TOTAL_IMAGENES;
   } // fin del método paintComponent

   // inicia la animación, o la reinicia si se vuelve a mostrar la ventana
   public void iniciarAnimacion()
   {
      if ( temporizadorAnimacion == null )
      {
         imagenActual = 0; // muestra la primera imagen

         // crea temporizador
         temporizadorAnimacion =
            new Timer( RETRASO_ANIMACION, new ManejadorTimer() );
```

Figura 21.2 | Animación de una serie de imágenes. (Parte 1 de 2).

```

            temporizadorAnimacion.start(); // inicia el objeto Timer
         } // fin de if
         else // temporizadorAnimacion ya existe; reinicia la animación
         {
            if ( ! temporizadorAnimacion.isRunning() )
               temporizadorAnimacion.restart();
         } // fin de else
      } // fin del método iniciarAnimacion

      // detiene el temporizador de la animación
      public void detenerAnimacion()
      {
         temporizadorAnimacion.stop();
      } // fin del método detenerAnimacion

      // devuelve el tamaño mínimo de la animación
      public Dimension getMinimumSize()
      {
         return getPreferredSize();
      } // fin del método getMinimumSize

      // devuelve el tamaño preferido de la animación
      public Dimension getPreferredSize()
      {
         return new Dimension( anchura, altura );
      } // fin del método getPreferredSize

      // clase interna para manejar los eventos de acción del objeto Timer
      private class ManejadorTimer implements ActionListener
      {
         // responde a un evento del objeto Timer
         public void actionPerformed( ActionEvent actionEvent )
         {
            repaint(); // vuelve a dibujar la animación
         } // fin del método actionPerformed
      } // fin de la clase ManejadorTimer
   } // fin de la clase AnimadorLogoJPanel
```

Figura 21.2 | Animación de una serie de imágenes. (Parte 2 de 2).

```
// Fig. 21.3: AnimadorLogo.java
// Animación de una serie de imágenes.
import javax.swing.JFrame;

public class AnimadorLogo
{
   // ejecuta la animación en un objeto JFrame
   public static void main( String args[] )
   {
      AnimadorLogoJPanel animacion = new AnimadorLogoJPanel();

      JFrame ventana = new JFrame( "Prueba de Animador" ); // establece la ventana
      ventana.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
      ventana.add( animacion ); // agrega el panel al marco

      ventana.pack();  // hace la ventana lo suficientemente grande para su GUI
```

Figura 21.3 | Visualización de imágenes en un objeto `JFrame`. (Parte 1 de 2).

```
17          ventana.setVisible( true );   // muestra la ventana
18
19          animacion.iniciarAnimacion();  // inicia la animación
20       } // fin de main
21    } // fin de la clase AnimadorLogo
```

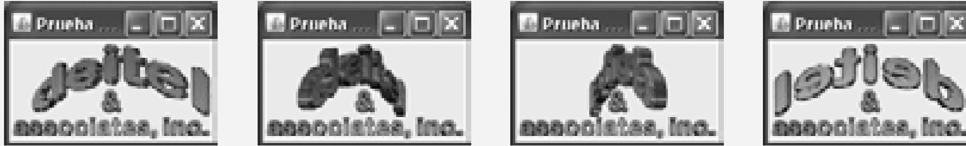


Figura 21.3 | Visualización de imágenes en un objeto `JFrame`. (Parte 2 de 2).

La clase `AnimadorLogoJPanel` (figura 21.2) mantiene un arreglo de objetos `ImageIcon` que se cargan en el constructor (líneas 24 a 36). En las líneas 29 a 31 se crea cada uno de los objetos `ImageIcon` y se cargan las 30 imágenes de la animación en el arreglo `imagenes`. El argumento del constructor utiliza la concatenación de cadenas para construir el nombre del archivo a partir de las piezas `"imagenes/"`, `NOMBRE_IMAGEN`, `cuenta` y `".gif"`. Cada una de las imágenes de la animación se encuentra en un archivo llamado `deitel#.gif`, en donde `#` es un valor en el rango de 0 a 29, el cual se especifica mediante la variable de control de ciclo `cuenta`. En las líneas 34 y 35 se determinan la anchura y altura de la animación, con base en el tamaño de la primera imagen del arreglo `imágenes`; suponemos que todas las imágenes tienen la misma anchura y altura.

Una vez que el constructor de `AnimadorLogoJPanel` carga las imágenes, el método `main` de la figura 21.3 establece la ventana en la que aparecerá la animación (líneas 12 a 17), y en la línea 19 se hace una llamada al método `iniciarAnimacion` de `AnimadorLogoJPanel` (declarado en las líneas 51 a 68 de la figura 21.2). Este método inicia la animación del programa por primera vez, o reinicia la animación que el programa detuvo anteriormente. [*Nota:* este método se llama cuando el programa se ejecuta por primera vez, para iniciar la animación. Aunque proporcionamos la funcionalidad para que este método reinicie la animación si se ha detenido, el ejemplo no llama al método para este propósito. Sin embargo, agregamos la funcionalidad en caso de que usted optara por agregar componentes de GUI que permitan al usuario iniciar y detener la animación]. Por ejemplo, para hacer que una animación sea "amigable para el usuario" en un applet, debe detenerse cuando el usuario cambie de páginas Web. Si el usuario regresa a la página Web con la animación, se puede llamar al método `iniciarAnimacion` para reiniciar la animación. La animación es controlada por una instancia de la clase `Timer` (del paquete `javax.swing`).

Un objeto `Timer` genera eventos `ActionEvent` en un intervalo fijo en milisegundos (que generalmente se especifica como argumento para el constructor de `Timer`) y notifica a todos sus objetos `ActionListener` cada vez que ocurre un evento `ActionEvent`. En la línea 53 se determina si la referencia `Timer` llamada `temporizadorAnimacion` es `null`. De ser así, se está llamando al método `iniciarAnimacion` por primera vez, y hay que crear un objeto `Timer` para que la animación pueda empezar. En la línea 55 se establece `imagenActual` en 0, lo cual indica que la animación debe empezar con la imagen que se encuentra en el primer elemento del arreglo `imagenes`. En las líneas 58 y 59 se asigna un nuevo objeto `Timer` a `temporizadorAnimacion`. El constructor de `Timer` recibe dos argumentos: el retraso en milisegundos (`RETRASO_ANIMACION` es 50, según lo especificado en la línea 17) y el objeto `ActionListener` que responderá a los objetos `ActionEvent` de `Timer`. Para el segundo argumento, se crea un objeto de la clase `ManejadorTimer`. Esta clase, que implementa a `ActionListener`, se declara en las líneas 89 a 96. En la línea 61 se inicia el objeto `Timer`. Una vez iniciado, `temporizadorAnimacion` generará un evento `ActionEvent` cada 50 milisegundos. Cada vez que se genera un evento `ActionEvent`, se hace una llamada al manejador de eventos `actionPerformed` de `Timer` (líneas 92 a 95). En la línea 94 se hace una llamada al método `repaint` de `AnimadorLogoJPanel` para programar una llamada al método `paintComponent` de `AnimadorLogoJPanel` (líneas 39 a 48). Recuerde que cualquier subclase de `JComponent` que dibuje, debe hacerlo en su método `paintComponent`. En el capítulo 11 vimos que la primera instrucción en cualquier método `paintComponent` debe ser una llamada al método `paintComponent` de la superclase, para asegurar que los componentes Swing se visualicen en forma correcta.

Si la animación empezó antes, entonces se creó nuestro objeto `Timer` y la condición en la línea 53 se evalúa como `false`. El programa continúa con las líneas 65 y 66, en donde se reinicia la animación que el programa

detuvo anteriormente. La condición `if` en la línea 65 utiliza el método `isRunning` de `Timer` para determinar si el objeto `Timer` está funcionando (es decir, si genera eventos). Si no está funcionando, en la línea 66 se hace una llamada al método `restart` de `Timer` para indicar que el objeto `Timer` debe empezar a generar eventos otra vez. Una vez que esto ocurre, se vuelve a llamar al método `actionPerformed` (el manejador de eventos del objeto `Timer`) en intervalos regulares. Cada vez, se realiza una llamada al método `repaint` (línea 94), lo cual provoca que se haga una llamada al método `paintComponent` y se muestra la siguiente imagen.

En la línea 43 se pinta el objeto `ImageIcon` almacenado en el elemento `imagenActual` del arreglo. En las líneas 46 y 47 se determina si el objeto `temporizadorAnimacion` está funcionando y, de ser así, se prepara la siguiente imagen a mostrar en pantalla, incrementando el valor de `imagenActual` en 1. El cálculo del residuo asegura que el valor de `imagenActual` esté en 0 (para repetir la secuencia de animación) cuando se incremente más allá de 29 (el índice del último elemento en el arreglo). La instrucción `if` asegura que, si se hace una llamada a `paintComponent` mientras el objeto `Timer` está detenido, se mostrará la misma imagen. Esto podría ser útil si se proporciona una GUI que permita al usuario iniciar y detener la animación. Por ejemplo, si la animación se detiene y el usuario la cubre con otra ventana, y después la descubre, se realizará una llamada al método `paint-Component`. En este caso, no queremos que la animación muestre la siguiente imagen (ya que la animación se ha detenido). Simplemente queremos que la ventana muestre la misma imagen hasta que se reinicie la animación.

El método `detenerAnimacion` (líneas 71 a 74) detiene la animación, llamando al método **`stop`** de `Timer` para indicar que el objeto `Timer` debe dejar de generar eventos. Esto evita que `actionPerformed` llame a `repaint` para iniciar el proceso de pintar la siguiente imagen en el arreglo. [*Nota:* al igual que para reiniciar la animación, este ejemplo define pero no utiliza el método `detenerAnimacion`. Hemos proporcionado este método para fines de demostración, o si el usuario desea modificar este ejemplo, de manera que pueda detener y reiniciar la animación].



Observación de ingeniería de software 21.1

Al crear una animación para utilizarla en un applet, debe proporcionar un mecanismo para deshabilitar la animación cuando el usuario navega en una nueva página Web, distinta de la página en la que reside el applet de la animación.

Recuerde que, al extender la clase `JPanel`, estamos creando un nuevo componente de GUI. Por ende, debemos asegurar que funcione igual que otros componentes para fines de usarlo en un esquema. Los administradores de esquemas a menudo utilizan el método **`getPreferredSize`** de un componente de GUI (heredado de la clase `java.awt.Component`) para determinar la anchura y altura preferidas del componente, al distribuirlo como parte de una GUI. Si un nuevo componente tiene una anchura y altura preferidas, debe sobrescribir el método `getPreferredSize` (líneas 83 a 86) para regresar esa anchura y altura como un objeto de la clase **`Dimension`** (paquete `java.awt`). La clase `Dimension` representa a la anchura y altura de un componente de GUI. En este ejemplo, las imágenes tienen 160 píxeles de anchura y 80 píxeles de largo, por lo que el método `getPreferredSize` devuelve un objeto `Dimension` que contiene los números 160 y 80 (determinados en las líneas 34 y 35).



Observación de apariencia visual 21.1

El tamaño predeterminado de un objeto `JPanel` es de 10 píxeles de ancho y 10 píxeles de alto.



Observación de apariencia visual 21.2

Al crear una subclase de `JPanel` (o de cualquier otra clase derivada de `JComponent`), sobrescriba el método `getPreferredSize` si el nuevo componente debe tener una anchura y altura preferidas específicas.

En las líneas 77 a 80 se sobrescribe el método **`getMinimumSize`**. Este método determina la anchura y altura mínimas del componente. Al igual que con el método `getPreferredSize`, los nuevos componentes deben sobrescribir el método `getMinimumSize` (también heredado de la clase `Component`). El método `getMinimumSize` simplemente llama a `getPreferredSize` (una práctica común de programación) para indicar que los tamaños mínimo y preferido son iguales. Algunos administradores de esquemas ignoran las dimensiones especificadas por estos métodos. Por ejemplo, las regiones `NORTH` y `SOUTH` de un objeto `BorderLayout` utilizan solamente la altura preferida del componente.



Observación de apariencia visual 21.3

Si un nuevo componente de GUI tiene una anchura y altura mínimas (es decir, medidas más pequeñas harían que el componente se desplegara en forma ineficiente), sobrescriba el método `getMinimumSize` *para regresar la anchura y altura mínimas como una instancia de la clase* `Dimension`.



Observación de apariencia visual 21.4

Para muchos componentes de GUI se implementa el método `getMinimumSize` *para devolver el resultado de una llamada al método* `getPreferredSize` *del componente.*

21.4 Mapas de imágenes

Los **mapas de imágenes** son una técnica común utilizada para crear páginas Web interactivas. Un mapa de imágenes es una imagen con **áreas activas** en las que el usuario puede hacer clic para realizar una tarea, como cargar una página Web distinta en un explorador Web. Cuando el usuario posiciona el puntero del ratón sobre un área activa, generalmente aparece un mensaje descriptivo en el área de estado del explorador Web, o en un cuadro de información sobre herramientas.

En la figura 21.4 se carga una imagen que contiene varios de los íconos de tips comunes que se utilizan en este libro. El programa permite al usuario posicionar el puntero del ratón sobre un icono y mostrar un mensaje descriptivo asociado con el icono. El manejador de eventos `mouseMoved` (líneas 39 a 43) toma las coordenadas del ratón y las pasa al método `traducirPosicion` (líneas 58 a 69). El método `traducirPosicion` evalúa las coordenadas para determinar sobre cuál icono se posicionó el ratón cuando ocurrió el **evento `mouseMoved`**; después el método devuelve un mensaje indicando lo que el icono representa. Este mensaje se muestra en la barra de estado del contenedor de applets, mediante el uso del método **`showStatus`** de la clase `Applet`.

Si hace clic en el applet de la figura 21.4, no se provocará ninguna acción. En el capítulo 24, Redes, hablamos sobre las técnicas requeridas para cargar otra página Web en un explorador Web mediante objetos `URL` y la interfaz `AppletContext`. Utilizando esas técnicas, este applet podría asociar cada icono con un objeto `URL` que el explorador Web mostraría cuando el usuario hiciera clic en el icono.

```
1  // Fig. 21.4: MapaImagenes.java
2  // Demostración de un mapa de imágenes.
3  import java.awt.event.MouseAdapter;
4  import java.awt.event.MouseEvent;
5  import java.awt.event.MouseMotionAdapter;
6  import java.awt.Graphics;
7  import javax.swing.ImageIcon;
8  import javax.swing.JApplet;
9
10 public class MapaImagenes extends JApplet
11 {
12    private ImageIcon imagenMapa;
13
14    private static final String leyendas[] = { "Error comun de programacion",
15       "Buena practica de programacion", "Observacion de apariencia visual",
16       "Tip de rendimiento", "Tip de portabilidad",
17       "Observacion de ingenieria de software", "Tip para prevenir errores" };
18
19    // establece los componentes de escucha del ratón
20    public void init()
21    {
22       addMouseListener(
23
24          new MouseAdapter() // clase interna anónima
25          {
26             // indica cuando el puntero del ratón sale del área del applet
```

Figura 21.4 | Mapa de imágenes. (Parte 1 de 3).

```
            public void mouseExited( MouseEvent evento )
            {
               showStatus( "Apuntador fuera de applet" );
            } // fin del método mouseExited
         } // fin de la clase interna anónima
      ); // fin de la llamada a addMouseListener

      addMouseMotionListener(

         new MouseMotionAdapter() // clase interna anónima
         {
            // determina el icono sobre el cual aparece el ratón
            public void mouseMoved( MouseEvent evento )
            {
               showStatus( traducirPosicion(
                  evento.getX(), evento.getY() ) );
            } // fin del método mouseMoved
         } // fin de la clase interna anónima
      ); // fin de la llamada a addMouseMotionListener

      imagenMapa = new ImageIcon( "iconos.png" ); // obtiene la imagen
   } // fin del método init

   // muestra imagenMapa
   public void paint( Graphics g )
   {
      super.paint( g );
      imagenMapa.paintIcon( this, g, 0, 0 );
   } // fin del método paint

   // devuelve leyenda del tip correspondiente, con base en las coordenadas del ratón
   public String traducirPosicion( int x, int y )
   {
      // si las coordenadas están fuera de la imagen, regresa de inmediato
      if ( x >= imagenMapa.getIconWidth() || y >= imagenMapa.getIconHeight() )
         return "";

      // determina el número del icono (0 - 6)
      double anchuraIcono = ( double ) imagenMapa.getIconWidth() / 7.0;
      int numeroIcono = ( int )( ( double ) x / anchuraIcono );

      return leyendas[ numeroIcono ]; // devuelve la leyenda del icono apropiado
   } // fin del método traducirPosicion
} // fin de la clase MapaImagenes
```

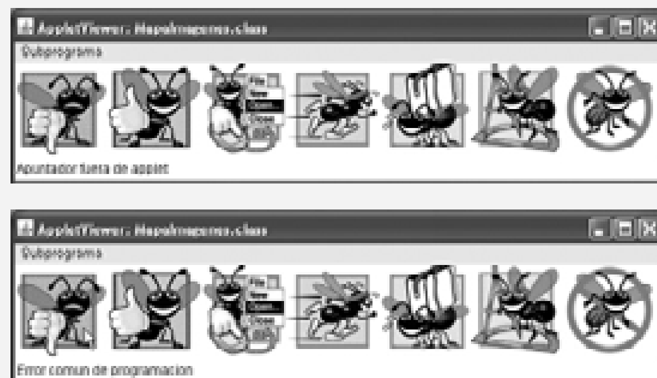


Figura 21.4 | Mapa de imágenes. (Parte 2 de 3).

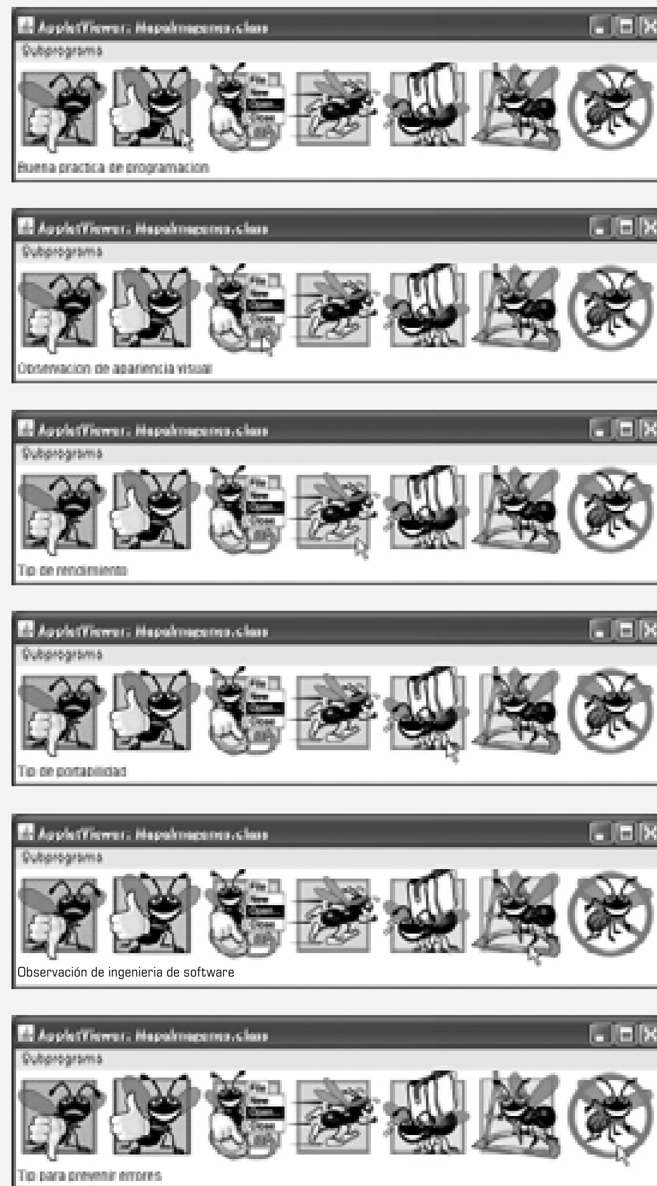


Figura 21.4 | Mapa de imágenes. (Parte 3 de 3).

21.5 Carga y reproducción de clips de audio

Los programas de Java pueden manipular y reproducir **clips de audio**. Los usuarios pueden capturar sus propios clips de audio, y hay muchos clips disponibles en productos de software y a través de Internet. Su sistema necesita estar equipado con hardware de audio (bocinas y una tarjeta de sonido) para poder reproducir los clips de audio.

Java proporciona varios mecanismos para reproducir sonidos en un applet. Los dos métodos más simples son el método **`play`** de `Applet` y el método **`play`** de la interfaz **`AudioClip`**. Hay varias capacidades de audio adicionales disponibles en el Marco de trabajo de medios de Java (Java Media Framework) y en las APIs de sonido de Java (Java Sound APIs). Si desea reproducir un sonido una vez en un programa, el método `play` de `Applet` carga

el sonido y lo reproduce una vez; el sonido se marca para la recolección de basura una vez que se reproduce. El método `play` de `Applet` tiene dos formas:

```
public void play( URL ubicacion, String nombreArchivoSonido );
public void play( URL urlSonido );
```

La primera versión carga el clip de audio almacenado en el archivo `nombreArchivoSonido` que se encuentra en `ubicacion` y reproduce el sonido. El primer argumento es normalmente una llamada al método `getDocumentBase` o **`getCodeBase`** del applet. El método `getDocumentBase` devuelve la ubicación del archivo HTML que cargó el applet. (Si el applet se encuentra en un paquete, el método devuelve la ubicación del paquete o archivo JAR que contiene a ese paquete). El método `getCodeBase` indica la ubicación del archivo `.class` del applet. La segunda versión del método `play` toma un objeto `URL` que contiene la ubicación y el nombre de archivo del audio clip. La instrucción

```
play( getDocumentBase(), "hi.au" );
```

carga el clip de audio que se encuentra en el archivo `hi.au` y reproduce el clip una vez.

El **motor de sonido** que reproduce los clips de audio soporta varios formatos de archivo de audio, incluyendo los formatos con extensión `.au` (**formato de archivo de audio de Sun**), `.wav` (**formato de archivo de onda de Windows**), `.aif` o `.aiff` (**formato de archivo AIFF de Macintosh**) y `.mid` o `.rmi` (**MIDI, Interfaz digital para instrumentos musicales**). El JMF (Marco de medios de Java) y las APIs de sonido de Java soportan formatos adicionales.

El programa de la figura 21.5 demuestra cómo cargar y reproducir un objeto `AudioClip` (paquete `java.applet`). Esta técnica es más flexible que el método `play` de `Applet`. Un applet puede utilizar un objeto `AudioClip` para almacenar audio que se use en repetidas ocasiones a lo largo de la ejecución de un programa. El método **`getAudioClip`** de `Applet` tiene dos formas que toman los mismos argumentos que el método `play` que se describió anteriormente. El método `getAudioClip` devuelve una referencia a un objeto `AudioClip`. Un objeto `AudioClip` tiene tres métodos: `play`, `loop` y `stop`. Como se mencionó antes, el método `play` reproduce el clip de audio una sola vez. El método **`loop`** reproduce continuamente el clip de audio en segundo plano. El método **`stop`** termina el clip de audio que se esté reproduciendo en ese momento. En el programa, cada uno de estos métodos se asocia con un botón en el applet.

Las líneas 62 y 63 en el método `init` del applet utilizan a `getAudioClip` para cargar dos archivos de audio: un archivo de onda de Windows (`welcome.wav`) y un archivo de audio de Sun (`hi.au`). El usuario puede seleccionar cuál clip de audio reproducir en el objeto `JComboBox` llamado `sonidoJComboBox`. Observe que el método `stop` del applet se sobrescribe en las líneas 68 a 71. Cuando el usuario cambia de páginas Web, el contenedor de applets llama al método `stop` del applet. Esto permite al applet dejar de reproducir el clip de audio. De no ser así, el clip de audio continúa reproduciéndose en segundo plano; incluso aunque el applet no se muestre en el navegador. Esto no es necesariamente un problema, pero puede ser molesto para el usuario que el clip de audio se reproduzca en forma continua. El método `stop` se proporciona aquí como una conveniencia para el usuario.

```
1  // Fig. 21.5: CargarAudioYReproducir.java
2  // Carga un clip de audio y lo reproduce.
3  import java.applet.AudioClip;
4  import java.awt.event.ItemListener;
5  import java.awt.event.ItemEvent;
6  import java.awt.event.ActionListener;
7  import java.awt.event.ActionEvent;
8  import java.awt.FlowLayout;
9  import javax.swing.JApplet;
10 import javax.swing.JButton;
11 import javax.swing.JComboBox;
12
13 public class CargarAudioYReproducir extends JApplet
14 {
```

Figura 21.5 | Carga y reproducción de un objeto `AudioClip`. (Parte 1 de 3).

```
   private AudioClip sonido1, sonido2, sonidoActual;
   private JButton reproducirJButton, continuoJButton, detenerJButton;
   private JComboBox sonidoJComboBox;

   // carga la imagen cuando el applet empieza a ejecutarse
   public void init()
   {
      setLayout( new FlowLayout() );

      String opciones[] = { "Welcome", "Hi" };
      sonidoJComboBox = new JComboBox( opciones ); // crea objeto JComboBox

      sonidoJComboBox.addItemListener(

         new ItemListener() // clase interna anónima
         {
            // detiene el sonido y lo cambia por la selección del usuario
            public void itemStateChanged( ItemEvent e )
            {
               sonidoActual.stop();
               sonidoActual = sonidoJComboBox.getSelectedIndex() == 0 ?
                  sonido1 : sonido2;
            } // fin del método itemStateChanged
         } // fin de la clase interna anónima
      ); // fin de la llamada al método addItemListener

      add( sonidoJComboBox ); // agrega objeto JComboBox al applet

      // establece el manejador de eventos de botón y los botones
      ManejadorBoton manejador = new ManejadorBoton();

      // crea objeto JButton Reproducir
      reproducirJButton = new JButton( "Reproducir" );
      reproducirJButton.addActionListener( manejador );
      add( reproducirJButton );

      // crea objeto JButton Continuo
      continuoJButton = new JButton( "Continuo" );
      continuoJButton.addActionListener( manejador );
      add( continuoJButton );

      // crea JButton Detener
      detenerJButton = new JButton( "Stop" );
      detenerJButton.addActionListener( manejador );
      add( detenerJButton );

      // carga los sonidos y establecet sonidoActual
      sonido1 = getAudioClip( getDocumentBase(), "welcome.wav" );
      sonido2 = getAudioClip( getDocumentBase(), "hi.au" );
      sonidoActual = sonido1;
   } // fin del método init

   // detiene el sonido cuando el usuario cambia a otra página Web
   public void stop()
   {
      sonidoActual.stop(); // detener AudioClip
   } // fin del método stop

   // clase interna privada para manejar eventos de botón
```

Figura 21.5 | Carga y reproducción de un objeto `AudioClip`. (Parte 2 de 3).

```
   private class ManejadorBoton implements ActionListener
   {
      // procesa los eventos de los botones reproducir, continuo y detener
      public void actionPerformed( ActionEvent actionEvent )
      {
         if ( actionEvent.getSource() == reproducirJButton )
            sonidoActual.play(); // reproducir AudioClip una vez
         else if ( actionEvent.getSource() == continuoJButton )
            sonidoActual.loop(); // reproducir AudioClip en forma continua
         else if ( actionEvent.getSource() == detenerJButton )
            sonidoActual.stop(); // detener AudioClip
      } // fin del método actionPerformed
   } // fin de la clase ManejadorBoton
} // fin de la clase CargarAudioYReproducir
```



Figura 21.5 | Carga y reproducción de un objeto `AudioClip`. (Parte 3 de 3).



Observación de apariencia visual 21.5

Al reproducir clips de audio en un applet o aplicación, debe proporcionar un mecanismo para que el usuario pueda deshabilitar el audio.

21.6 Reproducción de video y otros medios con el Marco de trabajo de medios de Java

Un video simple puede transmitir de manera concisa y eficiente una gran cantidad de información. Al reconocer el valor de incluir herramientas de multimedia extensibles en Java, Sun Microsystems, Intel y Silicon Graphics trabajaron en conjunto para producir la API del Marco de trabajo de medios de Java (JMF), que vimos brevemente en la sección 21.1. Mediante el uso de la API JMF, los programadores pueden crear aplicaciones en Java para reproducir, editar, transmitir y capturar muchos tipos de medios populares. Mientras que las características de la API JMF son bastante extensas, en esta sección introduciremos brevemente algunos formatos de medios populares, y demostraremos cómo reproducir video mediante el uso de la API JMF.

IBM y Sun desarrollaron la especificación más reciente de la API JMF: la versión 2.0. Sun también proporciona una implementación de referencia de la especificación de la API JMF (JMF 2.1.1e), la cual soporta tipos de archivos de medios como `.avi` (**Microsoft Audio/Video Interleave**), `.swf` (**películas de Macromedia Flash 2**), `.spl` (**Future Splash**), `.mp3` (**Audio MPEG nivel 3**), `.mid` o `.midi` (MIDI; Interfaz digital de instrumentos musicales), `.mpeg` y `.mpg` (**videos MPEG-1**), `.mov` (**QuickTime**), `.au` (formato de archivo Sun Audio) y `.aif` o `.aiff` (formato de archivo Macintosh AIFF). Ya hemos visto algunos de estos tipos de archivos.

En la actualidad, la API JMF está disponible como una extensión separada del Kit de desarrollo de software para Java 2. La implementación más reciente de la API JMF (2.1.1e) se puede descargar de:

`java.sun.com/products/java-media/jmf/2.1.1/download.html`

Necesita aceptar el contrato de licencia antes de descargar el archivo.

El sitio Web JMF proporciona versiones de la API JMF que aprovechan las características de rendimiento de ciertas plataformas. Por ejemplo, el JMF Windows Performance Pack proporciona un extenso soporte para medios y dispositivos, para los programas de Java que se ejecutan en plataformas Microsoft Windows. El sitio Web oficial de la API JMF (`java.sun.com/products/java-media/jmf`) proporciona un soporte que se actualiza en forma continua, información y recursos para los programadores de la API JMF.

Una vez que el archivo termine de descargarse, ábralo y siga las instrucciones en pantalla para instalar el programa. Deje todas las opciones predeterminadas. Tal vez necesite reiniciar su equipo para terminar la instalación.

Creación de un reproductor de medios simple

La API JMF ofrece varios mecanismos para reproducir medios. El mecanismo más simple es usar objetos que implementen a la interfaz **`Player`**, declarada en el paquete **`javax.media`**. El paquete **`javax.media`** y sus subpaquetes contienen las clases que componen el Marco de trabajo de medios de Java. Para reproducir un clip de medios, primero debe crear un objeto `URL` que haga referencia a él. Después debe pasar el `URL` como argumento para el método `static` **`createRealizedPlayer`** de la clase `Manager`, para obtener un objeto `Player` para el clip de medios. La clase **`Manager`** declara métodos utilitarios para acceder a los recursos del sistema para reproducir y manipular medios. En la figura 21.6 se declara un objeto `JPanel` que demuestra algunos de estos métodos.

```
// Fig 21.6: PanelMedios.java
// Un objeto JPanel que reproduce medios de un URL
import java.awt.BorderLayout;
import java.awt.Component;
import java.io.IOException;
import java.net.URL;
import javax.media.CannotRealizeException;
import javax.media.Manager;
import javax.media.NoPlayerException;
import javax.media.Player;
import javax.swing.JPanel;

public class PanelMedios extends JPanel
{
   public PanelMedios( URL urlMedios )
   {
      setLayout( new BorderLayout() ); // usa un objeto BorderLayout

      // Usa componentes ligeros para compatibilidad con Swing
      Manager.setHint( Manager.LIGHTWEIGHT_RENDERER, true );

      try
      {
         // crea un objeto Player para reproducir los medios especificados en el URL
         Player reproductorMedios = Manager.createRealizedPlayer( urlMedios );

         // obtiene los componentes para el video y los controles de reproducción
         Component video = reproductorMedios.getVisualComponent();
         Component controles = reproductorMedios.getControlPanelComponent();

         if ( video != null )
            add( video, BorderLayout.CENTER ); // agrega el componente de video

         if ( controles != null )
            add( controles, BorderLayout.SOUTH ); // agrega los controles

         reproductorMedios.start(); // empieza a reproducir el clip de medios
      } // fin de try
      catch ( NoPlayerException noPlayerException )
      {
         System.err.println( "No se encontro reproductor de medios" );
      } // fin de catch
      catch ( CannotRealizeException cannotRealizeException )
      {
         System.err.println( "No se pudo realizar el reproductor de medios" );
      } // fin de catch
      catch ( IOException iOException )
      {
```

Figura 21.6 | Objeto `JPanel` que reproduce un archivo de medios de un URL. (Parte 1 de 2).

```
49           System.err.println( "Error al leer del origen" );
50        } // fin de catch
51     } // fin del constructor de PanelMedios
52  } // fin de la clase PanelMedios
```

Figura 21.6 | Objeto `JPanel` que reproduce un archivo de medios de un URL. (Parte 2 de 2).

El constructor (líneas 15 a 51) establece el objeto `JPanel` para reproducir el archivo de medios especificado por el parámetro `URL`. `PanelMedios` utiliza un `BorderLayout` (línea 17). En la línea 20 se invoca al método `static` **`setHint`** para establecer la bandera **`Manager.LIGHTWEIGHT_RENDERER`** en `true`. Esto indica al objeto `Manager` que debe usar un renderizador ligero que sea compatible con los componentes ligeros de Swing, en contraste al renderizador pesado predeterminado. Dentro del bloque `try` (líneas 22 a 38), en la línea 25 se invoca el método `static createRealizedPlayer` de la clase `Manager`, para crear y realizar un objeto `Player` que reproduzca el archivo de medios. Cuando se realiza un objeto `Player`, identifica los recursos del sistema que necesita para reproducir los medios. Dependiendo del archivo, la realización puede ser un proceso que consume muchos recursos y tiempo. El método `createRealizedPlayer` lanza tres excepciones verificadas, **`NoPlayerException`**, **`CannotRealizeException`** e `IOException`. Una excepción `NoPlayerException` indica que el sistema no pudo encontrar un reproductor para el formato del archivo. Una excepción `CannotRealizeException` indica que el sistema no pudo identificar apropiadamente los recursos que necesita un archivo de medios. Una excepción `IOException` indica que hubo un error al leer el archivo. Estas excepciones se manejan en el bloque `catch` de las líneas 39 a 50.

En la línea 28 se invoca el método **`getVisualComponent`** de `Player` para obtener un objeto `Component` que muestre el aspecto visual (por lo general, video) del archivo de medios. En la línea 29 se invoca el método **`getControlPanelComponent`** de `Player` para obtener un objeto `Component` que proporcione controles de reproducción y medios. Estos componentes se asignan a las variables locales `video` y `controles`, respectivamente. La instrucción `if` en las líneas 31 y 32, y en las líneas 34 y 35 agregan los objetos `video` y `controles`, si existen. El objeto `Component` llamado `video` se agrega a la región `CENTER` (línea 32), para llenar cualquier espacio disponible en el objeto `JPanel`. El objeto `Component` llamado `controles`, que se agrega a la región `SOUTH`, por lo general, proporciona los siguientes controles:

1. Un control deslizable de posicionamiento, para saltar a ciertos puntos en el clip de medios.
2. Un botón de pausa.
3. Un botón de volumen, que proporciona el control del volumen al hacer clic con el botón derecho del ratón, y una función de silencio (muting) al hacer clic con el botón izquierdo.
4. Un botón de propiedades de medios, que proporciona información detallada sobre los medios al hacer clic con el botón izquierdo del ratón, y un control de la velocidad de trama al hacer clic con el botón derecho.

En la línea 37 se llama al método **`start`** de `Player` para empezar a reproducir el archivo de medios. En las líneas 39 a 50 se manejan las diversas excepciones que lanza `createRealizedPlayer`.

La aplicación en la figura 21.7 muestra un cuadro de diálogo `JFileChooser` para que el usuario seleccione un archivo de medios. Después crea un objeto `PanelMedios` que reproduce el archivo seleccionado y crea un objeto `JFrame` para mostrar el objeto `PanelMedios`.

```
// Fig. 21.7: PruebaMedios.java
// Un reproductor de medios simple
import java.io.File;
import java.net.MalformedURLException;
import java.net.URL;
```

Figura 21.7 | Aplicación de prueba que crea un objeto `PanelMedios` a partir de un archivo seleccionado por el usuario. (Parte 1 de 3).

```
import javax.swing.JFileChooser;
import javax.swing.JFrame;

public class PruebaMedios
{
   // inicia la aplicación
   public static void main( String args[] )
   {
      // crea un selector de archivo
      JFileChooser selectorArchivo = new JFileChooser();

      // muestra cuadro de diálogo para abrir archivo
      int resultado = selectorArchivo.showOpenDialog( null );

      if ( resultado == JFileChooser.APPROVE_OPTION ) // el usuario eligió un archivo
      {
         URL urlMedios = null;

         try
         {
            // obtiene el archivo como un URL
            urlMedios = selectorArchivo.getSelectedFile().toURL();
         } // fin de try
         catch ( MalformedURLException malformedURLException )
         {
            System.err.println( "No se pudo crear URL para el archivo" );
         } // fin de catch

         if ( urlMedios != null ) // sólo lo muestra si hay un URL válido
         {
            JFrame pruebaMedios = new JFrame( "Probador de medios" );
            pruebaMedios.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );

            PanelMedios panelMedios = new PanelMedios( urlMedios );
            pruebaMedios.add( panelMedios );

            pruebaMedios.setSize( 300, 300 );
            pruebaMedios.setVisible( true );
         } // fin de if interior
      } // fin de if exterior
   } // fin de main
} // fin de la clase PruebaMedios
```

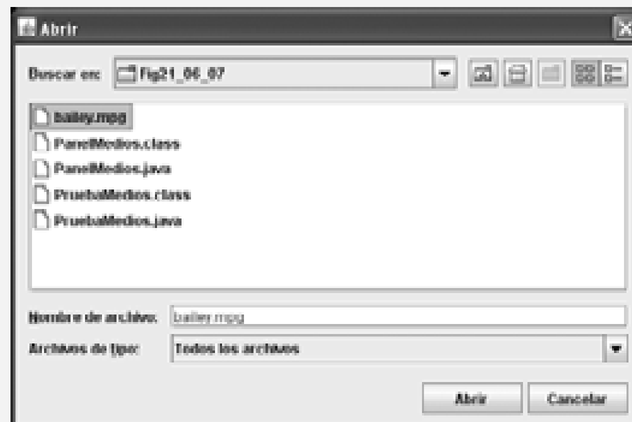


Figura 21.7 | Aplicación de prueba que crea un objeto `PanelMedios` a partir de un archivo seleccionado por el usuario. (Parte 2 de 3).



Figura 21.7 | Aplicación de prueba que crea un objeto `PanelMedios` a partir de un archivo seleccionado por el usuario. (Parte 3 de 3).

El método `main` (líneas 12 a 46) asigna un nuevo objeto `JFileChooser` a la variable local `selectorArchivo` (línea 15), muestra un cuadro de diálogo para abrir un archivo (línea 18) y asigna el valor de retorno a `resultado`. En la línea 20 se comprueba `resultado` para determinar si el usuario eligió un archivo. Para crear un objeto `Player` y reproducir el archivo de medios seleccionado, debe convertir el objeto `File` devuelto por `JFileChooser` en un objeto `URL`. El método **`toURL`** de la clase `File` devuelve un `URL` que apunta al objeto `File` en el sistema, con la posibilidad de lanzar una excepción `MalformedURLException` si no puede crear un objeto `URL` para el objeto `File`. La instrucción `try` (líneas 24 a 32) crea un objeto `URL` para el archivo seleccionado y lo asigna a `urlMedios`. La instrucción `if` en las líneas 34 a 44 comprueba que `urlMedios` no sea `null` y crea los componentes de GUI para reproducir los medios.

21.7 Conclusión

En este capítulo aprendió a crear aplicaciones más excitantes, al incluir sonido, imágenes, gráficos y video. Presentamos las herramientas de multimedia de Java, incluyendo la API del Marco de trabajo de medios de Java, la API de Sonido de Java y la API Java 3D. Utilizó las clases `Image` e `ImageIcon` para mostrar y manipular imágenes almacenadas en archivos, y aprendió acerca de los distintos formatos de imágenes en un orden específico. Utilizó mapas de imágenes para hacer que una aplicación tenga más interactividad. Después aprendió a cargar clips de audio, y a reproducirlos una vez o en un ciclo continuo. El capítulo concluyó con una demostración de cómo cargar y reproducir video. En el siguiente capítulo, continuará su estudio sobre los conceptos de GUI, partiendo de las técnicas que aprendió en el capítulo 11.

21.8 Recursos Web

`www.nasa.gov/multimedia/highlights/index.html`
La galería *NASA Multimedia Gallery (Galería de multimedios de NASA)* contiene una amplia variedad de imágenes, clips de audio y de video que puede descargar y utilizar para probar sus programas de multimedia de Java.

`www.anbg.gov.au/anbg/index.html`
El sitio Web *Australian National Botanic Gardens (Jardines botánicos nacionales australianos)* proporciona vínculos a los sonidos de muchos animales. Por ejemplo, pruebe el vínculo *Common Birds* (*Aves comunes*) bajo la sección "Animals in the Gardens" (Animales en los jardines).

`www.thefreesite.com`
Este sitio tiene vínculos a sonidos e imágenes prediseñadas gratuitas.

`www.soundcentral.com`
SoundCentral proporciona clips de audio en los formatos WAV, AU, AIFF y MIDI.

`www.animationfactory.com`
El sitio *Animation Factory (Fábrica de animaciones)* proporciona miles de animaciones GIF gratuitas para uso personal.

`www.clipart.com`
ClipArt.com es un servicio basado en suscripciones, que ofrece imágenes y sonidos.

`www.pngart.com`
PNGART.com proporciona cerca de 50,000 imágenes gratuitas en formato PNG.

`java.sun.com/developer/techDocs/hi/repository`
El sitio *Java look and feel Graphics Repository (Depósito de gráficos de apariencia visual de Java)* proporciona imágenes diseñadas para utilizarse en una GUI de Swing, incluyendo imágenes de botones de barras de herramientas.

`www.freebyte.com/graphicprograms/`
Esta guía contiene vínculos a varios programas de software de gráficos gratuitos. El software se puede utilizar para modificar imágenes y dibujar gráficos.

`graphicssoft.about.com/od/pixelbasedfreewin/`
Este sitio Web proporciona vínculos para programas de gráficos gratuitos, diseñados para usarse en equipos Windows.

Referencias de la API de multimedia de Java

`java.sun.com/products/java-media/jmf/`
Ésta es la página de inicio del *Marco de trabajo de medios de Java (JMF)*. Aquí puede descargar la implementación de Sun más reciente de la JMF. Este sitio también contiene la documentación para la JMF.

`java.sun.com/products/java-media/sound/`
La página de inicio de la *API de Sonido de Java*. Esta API proporciona herramientas para reproducir y grabar audio.

`java3d.dev.java.net/`
La página de inicio de la *API Java 3D*. Esta API se puede utilizar para producir imágenes tridimensionales, típicas de los videojuegos actuales.

`java.sun.com/developer/onlineTraining/java3d/`
Este sitio proporciona un tutorial sobre la *API Java 3D*.

`java.sun.com/products/java-media/jai/`
La página de inicio de la *API de Manipulación avanzada de imágenes de Java*. Esta API proporciona herramientas de procesamiento de imágenes, como la mejora del contraste, recorte, escalado y deformación (warping) geométrica.

`java.sun.com/products/java-media/speech/`
La *API de Reconocimiento de voz de Java* permite a los programas realizar la síntesis y el reconocimiento de voz.

`freetts.sourceforge.net/docs/index.php`
FreeTTS es una implementación de la API de Reconocimiento de voz de Java.

`java.sun.com/products/java-media/2D/`
Ésta es la página de inicio de la *API Java 2D*. Esta API (presentada en el capítulo 12) proporciona herramientas para gráficos bidimensionales complejos.

`java.sun.com/javase/6/docs/technotes/guides/imageio/index.html`
Este sitio contiene una guía para la *API de E/S de imágenes de Java*, la cual permite a los programas cargar y guardar imágenes, usando formatos que las APIs de Java no soportan actualmente.

Resumen

Sección 21.2 Cómo cargar, mostrar y escalar imágenes

- El método `getImage` de `Applet` carga un objeto `Image`.
- El método `getDocumentBase` de `Applet` devuelve la ubicación del archivo HTML del applet en Internet, como un objeto de la clase `URL`.
- Java soporta varios formatos de imagen, incluyendo GIF (Formato de intercambio de gráficos), JPEG (Grupo unido de expertos en fotografía) y PNG (Gráficos portables de red). Los nombres de archivo para estos tipos terminan con `.gif`, `.jpg` (o `.jpeg`) y `.png`, respectivamente.
- La clase `ImageIcon` proporciona constructores que permiten inicializar un objeto `ImageIcon` con una imagen del equipo local, o almacenadas en un servidor Web en Internet.
- El método `drawImage` de `Graphics` acepta cuatro argumentos: una referencia al objeto `Image` en el que se almacena la imagen, las coordenadas *x* y *y* en donde debe mostrarse la imagen y una referencia a un objeto `ImageObserver`.

- Otra versión del método `drawImage` de `Graphics` imprime en pantalla una imagen escalada. Los argumentos cuarto y quinto especifican la anchura y la altura de la imagen, para fines de mostrarla en pantalla.
- La interfaz `ImageObserver` se implementa mediante la clase `Component`. Los objetos `ImageObserver` reciben notificaciones a medida que se carga un objeto `Image` y se actualiza en pantalla, en caso de que no haya estado completo al momento de mostrarlo.
- El método `paintIcon` de `ImageIcon` muestra la imagen del objeto `ImageIcon`. El método requiere cuatro argumentos: una referencia al objeto `Component` en el que se mostrará la imagen, una referencia al objeto `Graphics` utilizado para desplegar (render) la imagen, la coordenada *x* de la esquina superior izquierda de la imagen y la coordenada *y* de la esquina superior izquierda.
- Un objeto `URL` representa a un Localizador uniforme de recursos, el cual es un apuntador a un recurso en World Wide Web, en su equipo o en cualquier equipo conectado en red.

Sección 21.3 Animación de una serie de imágenes

- Los objetos `Timer` generan eventos `ActionEvent` en intervalos de milisegundos fijos. El constructor de `Timer` recibe un retraso en milisegundos y un objeto `ActionListener`. El método `start` de `Timer` inicia el objeto `Timer`. El método `stop` indica que el objeto `Timer` debe dejar de generar eventos. El método `restart` indica que el objeto `Timer` debe empezar a generar eventos de nuevo.

Sección 21.4 Mapas de imágenes

- Un mapa de imágenes es una imagen que tiene áreas activas en las que el usuario puede hacer clic para realizar una tarea, como cargar una página Web distinta en un explorador Web.

Sección 21.5 Carga y reproducción de clips de audio

- El método `play` de `Applet` tiene dos formas:

```
public void play ( URL ubicacion, String nombreArchivoSonido );
public void play ( URL urlSonido );
```

  Una versión carga desde `ubicacion` el clip de audio almacenado en el archivo `nombreArchivoSonido` y lo reproduce. La otra versión recibe un URL que contiene la ubicación y el nombre de archivo del clip de audio.
- El método `getDocumentBase` de `Applet` indica la ubicación del archivo HTML que cargó el applet. El método `getCodeBase` indica en dónde se encuentra el archivo `.class` para un applet.
- El motor de sonido que reproduce clips de audio soporta varios formatos de archivo de audio, incluyendo `.au` (formato de archivo Sun Audio), `.wav` (formato de archivo Windows Wave), `.aif` o `.aiff` (formato de archivo Macintosh AIFF) y `.mid` o `.rmi` (formato de archivo de Interfaz digital de instrumentos musicales, MIDI). El Marco de trabajo de medios de Java (JMF) soporta formatos adicionales.
- El método `getAudioClip` de `Applet` tiene dos formas que reciben los mismos argumentos que el método `play`. El método `getAudioClip` devuelve una referencia a un objeto `AudioClip`. Los objetos `AudioClip` tienen tres métodos: `play`, `loop` y `stop`. El método `play` reproduce el clip de audio sólo una vez. El método `loop` reproduce en forma continua el clip de audio. El método `stop` termina un clip de audio que se esté reproduciendo en un momento dado.

Sección 21.6 Reproducción de video y otros medios con el Marco de trabajo de medios de Java

- Sun Microsystems, Intel y Silicon Graphics trabajaron en conjunto para producir el Marco de trabajo de medios de Java (JMF).
- El paquete `javax.media` y sus subpaquetes contienen las clases que componen el Marco de trabajo de medios de Java.
- La clase `Manager` declara métodos utilitarios para acceder a los recursos del sistema para reproducir y manipular medios.
- El método `toURL` de la clase `File` devuelve un `URL` que apunta al objeto `File` en el sistema.

Terminología

`.aif`, extensión de archivo
`.aiff`, extensión de archivo
`.au`, extensión de archivo
`.avi`, extensión de archivo
`.gif`, extensión de archivo
`.jpeg`, extensión de archivo
`.jpg`, extensión de archivo
`.mid`, extensión de archivo
`.mov`, extensión de archivo
`.mp3`, extensión de archivo

.mpeg, extensión de archivo
.mpg, extensión de archivo
.png, extensión de archivo
.rmi, extensión de archivo
.spl, extensión de archivo
.swf, extensión de archivo
.wav, extensión de archivo
área activa
AudioClip, interfaz
CannotRealizePlayerException, excepción
clip de audio
createRealizedPlayer, método de la clase Manager
Dimension, clase
drawImage, método de la clase Graphics
E/S de imágenes de Java, API
Formato de intercambio de gráficos (GIF)
Future Splash (.spl), archivos
getAudioClip, método de la clase Applet
getCodeBase, método de la clase Applet
getControlPanelComponent, método de la interfaz Player
getDocumentBase, método de la clase Applet
getImage, método de la clase Applet
getMinimumSize, método de la clase Component
getPreferredSize, método de la clase Component
getVisualComponent, método de la interfaz Player
Gráficos portables de red (PNG)
Graphics, clase
Grupo unido de expertos en fotografía (JPEG)
Image, clase
ImageIcon, clase
ImageObserver, interfaz
Interfaz digital de instrumentos musicales (MIDI), formato de archivo (extensiones .mid o .rmi)
Java 3D, API
javax.media, paquete
LIGHTWEIGHT_RENDERER, constante de la clase Manager
loop, método de la interfaz AudioClip
Macintosh AIFF, formato de archivo (extensiones .aif o .aiff)
Macromedia Flash 2, películas (.swf)
Manager, clase
Manipulación avanzada de imágenes de Java, API
mapa de imágenes
Marco de trabajo de medios de Java (JMF), API
Microsoft Audio/Video Interleave (.avi), archivo
motor de sonido
MPEG Nivel 3, archivos de audio (.mp3)
MPEG-1, videos (.mpeg, .mpg)
multimedia
NoPlayerException, excepción
paintIcon, método de la clase ImageIcon
play, método de la clase Applet
play, método de la interfaz AudioClip
Player, interfaz
QuickTime (.mov), archivos
setHint, método de la clase Manager
showStatus, método de la clase Applet
Síntesis de voz de Java, API
sonido
Sonido de Java, API
start, método de la interfaz Player
stop, método de la clase Timer
stop, método de la interfaz AudioClip
Sun Audio, formato de archivo (extensión .au)
toURL, método de la clase File
URL, clase
video
Windows Wave, formato de archivo (extensión .wav)

Ejercicios de autoevaluación

21.1 Complete las siguientes oraciones:

a) El método __________ de Applet carga una imagen en un applet.
b) El método __________ de Graphics muestra una imagen en un applet.
c) Java proporciona dos mecanismos para reproducir sonidos en un applet: el método play de Applet y el método play de la interfaz __________.
d) Un __________ es una imagen que tiene áreas activas, en las que el usuario puede hacer clic para realizar una tarea, como cargar una página Web distinta.
e) El método __________ de la clase ImageIcon muestra la imagen del objeto ImageIcon.
f) Java soporta varios formatos de imagen, incluyendo __________, __________ y __________.

21.2 Conteste con *verdadero* o *falso* a cada una de las siguientes proposiciones; en caso de ser *falso*, explique por qué.

a) Un sonido será recolectado como basura tan pronto como haya dejado de reproducirse.
b) La clase ImageIcon proporciona constructores que permiten a un objeto ImageIcon ser inicializado con sólo una imagen proveniente del equipo local.
c) El método play de la clase AudioClip reproduce en forma continua un clip de audio.
d) La API de E/S de imágenes de Java se utiliza para agregar gráficos en 3D a una aplicación de Java.

e) El método `getDocumentBase` de `Applet` devuelve, como un objeto de la clase `URL`, la ubicación en Internet del archivo HTML que invocó al applet.

Respuestas a los ejercicios de autoevaluación

21.1 a) `getImage`. b) `drawImage`. c) `AudioClip`. d) mapa de imágenes. e) `paintIcon`. f) Formato de intercambio de gráficos (GIF), Grupo unido de expertos en fotografía (JPEG), Gráficos portables de red (PNG).

21.2 a) Verdadero. b) Falso. `ImageIcon` puede cargar imágenes de Internet también. c) Falso. El método `play` de la clase `AudioClip` reproduce un clip de audio sólo una vez. El método `loop` de la clase `AudioClip` reproduce en forma continua un clip de audio. d) Falso. La API Java 3D se utiliza para crear y modificar gráficos en 3D. La API de E/S de imágenes de Java se utiliza para leer y escribir imágenes en archivos. e) Verdadero.

Ejercicios

21.3 Describa cómo hacer que una animación sea "amigable para el explorador Web".

21.4 Describa los métodos de Java para reproducir y manipular clips de audio.

21.5 Explique cómo se utilizan los mapas de imágenes. Enliste varios ejemplos de su uso.

21.6 *(Borrar una imagen al azar)* Suponga que se muestra una imagen en un área rectangular de la pantalla. Una manera de borrar la imagen es simplemente asignar a cada píxel el mismo color inmediatamente, pero éste es un efecto visual torpe. Escriba un programa en Java para mostrar una imagen y luego borrarla, utilizando la generación de números aleatorios para seleccionar píxeles individuales y borrarlos. Una vez que se haya borrado la mayor parte de la imagen, borre el resto de los píxeles al mismo tiempo. Puede dibujar píxeles individuales como una línea que inicie y termine en las mismas coordenadas. Tal vez sea conveniente que pruebe distintas variantes de este problema. Por ejemplo, podría mostrar líneas o figuras al azar, para borrar regiones de la pantalla.

21.7 *(Texto intermitente)* Cree un programa en Java para mostrar repetidamente texto intermitente en la pantalla. Para ello, alterne el texto con una imagen de fondo simple a color. Permita al usuario controlar la "velocidad de parpadeo" y el color o patrón de fondo. Necesitará usar los métodos **`getDelay`** y **`setDelay`** de la clase `Timer`. Estos métodos se utilizan para recuperar y establecer el intervalo en milisegundos entre `ActionEvents`, respectivamente.

21.8 *(Imagen intermitente)* Cree un programa en Java para mostrar repetidamente una imagen intermitente en la pantalla. Para ello, alterne la imagen con una imagen de fondo simple a color.

21.9 *(Reloj digital)* Implemente un programa para mostrar un reloj digital en la pantalla.

21.10 *(Atraer la atención hacia una imagen)* Si desea enfatizar una imagen, puede colocar una fila de bombillas simuladas alrededor de su imagen. Puede dejar que las bombillas parpadeen al unísono, o puede dejar que se prendan y apaguen en secuencia, una después de otra.

21.11 *(Acercamiento/alejamiento de imagen)* Cree un programa que le permita acercarse o alejarse de una imagen.

Sección especial: proyectos de multimedia retadores

Los ejercicios anteriores están adaptados al texto y diseñados para evaluar la comprensión del lector en cuanto a los conceptos fundamentales de multimedia. En esta sección incluimos una colección de proyectos avanzados de multimedia. El lector encontrará que estos problemas son retadores, pero interesantes. Los problemas varían considerablemente en cuanto al grado de dificultad. Algunos requieren de una o dos horas para escribir e implementar el programa correspondiente. Otros son útiles como tareas de laboratorio, que podrían requerir de dos o tres semanas de estudio e implementación. Algunos son proyectos finales retadores. (*Nota para los instructores:* No se proporcionan las soluciones para estos ejercicios).

21.12 *(Animación)* Cree un programa de animación en Java, de propósito general. Su programa deberá permitir al usuario especificar la secuencia de cuadros a mostrar, la velocidad a la que se van a mostrar las imágenes, los clips de audio que deberán reproducirse mientras la animación se ejecuta, y así sucesivamente.

21.13 *(Quintillas)* Modifique el programa para escribir quintillas del ejercicio 10.10, para que cante las quintillas que su programa cree.

21.14 *(Transición aleatoria entre imágenes)* Este proyecto proporciona un agradable efecto visual. Si va a mostrar una imagen en cierta área de la pantalla y desea cambiar a otra imagen en la misma área de la pantalla, almacene la nueva

imagen de pantalla en un búfer fuera de la pantalla, y copie al azar píxeles de la nueva imagen hacia el área que se va a mostrar en pantalla, cubriendo los píxeles anteriores en esas posiciones. Cuando se haya copiado la mayor parte de los píxeles, copie toda la nueva imagen en el área que se va a mostrar en pantalla, para asegurarse de estar mostrando la nueva imagen completa. Para implementar este programa, tal vez necesite usar las clases `PixelGrabber` y `MemoryImageSource` (consulte la documentación de las APIs de Java para obtener las descripciones de esas clases). Tal vez sea conveniente que pruebe distintas variantes de este problema. Por ejemplo, trate de seleccionar todos los píxeles en una línea recta o figura seleccionada al azar en la nueva imagen, y cubra esos píxeles por encima de las posiciones correspondientes de la imagen anterior.

21.15 *(Audio de fondo)* Agregue audio de fondo a una de sus aplicaciones favoritas, utilizando el método `loop` de la clase `AudioClip` para reproducir el sonido en segundo plano mientras interactúa con su aplicación en forma normal.

21.16 *(Marquesina desplazable)* Cree un programa en Java para desplazar caracteres punteados de derecha a izquierda (o de izquierda a derecha, si es apropiado para su lenguaje), a lo largo de un letrero tipo marquesina. Como una opción, muestre el texto en un ciclo continuo de manera que el texto desaparezca en un extremo y reaparezca en el otro.

21.17 *(Marquesina con imagen desplazable)* Cree un programa en Java para mostrar una imagen a lo largo de una pantalla tipo marquesina.

21.18 *(Reloj análogo)* Cree un programa en Java para mostrar un reloj análogo con manecillas para las horas, minutos y segundos que se desplacen apropiadamente, a medida que vaya cambiando la hora.

21.19 *(Audio dinámico y calidoscopio gráfico)* Escriba un programa de calidoscopio que muestre gráficos reflejados para simular el popular juguete para niños. Incorpore efectos de audio que "reflejen" los gráficos de su programa que cambian en forma dinámica.

21.20 *(Generador automático de rompecabezas)* Cree un generador y manipulador de rompecabezas en Java. El usuario especifica una imagen. Su programa carga y muestra la imagen, y después divide la imagen en varias figuras seleccionadas al azar, y las revuelve. El usuario entonces utiliza el ratón para desplazar las piezas alrededor y tratar de resolver el rompecabezas. Agregue sonidos de audio apropiados a medida que se vayan moviendo las piezas y se inserten en el lugar correcto. Tal vez sea conveniente tener etiquetas en cada pieza y en el lugar en el que pertenece; después se pueden utilizar efectos de audio para ayudar al usuario a obtener las piezas en las posiciones correctas.

21.21 *(Programa para generar y recorrer laberintos)* Desarrolle un programa basado en multimedia para generar laberintos y recorrerlos, con base en los programas de laberintos que escribió en los ejercicios 15.20 a 15.22. Deje que el usuario personalice el laberinto, especificando el número de filas y columnas junto con el nivel de dificultad. Haga que un ratón animado recorra el laberinto. Utilice audio para dramatizar el movimiento de su personaje ratón.

21.22 *(Bandido de un brazo)* Desarrolle una simulación multimedia de una máquina tragamonedas, conocida como "bandido de un brazo". Debe tener tres ruedas giratorias. Coloque varias frutas y símbolos en cada rueda. Use la generación de números aleatorios para simular los giros de cada rueda y la acción de detenerse en un símbolo.

21.23 *(Carrera de caballos)* Cree una simulación en Java de una carrera de caballos. Debe tener varios competidores. Use clips de audio para tener un anunciador. Reproduzca los clips de audio apropiados para indicar el estado correcto de cada uno de los competidores, a lo largo de la carrera. Use clips de audio para anunciar el resultado final. Sería conveniente que tratara de simular el tipo de juegos de carreras de caballos que se juegan en los carnavales. Los jugadores toman turnos en el ratón y tienen que realizar ciertas manipulaciones hábiles con el ratón para que sus caballos avancen.

21.24 *(Juego de tejo)* Desarrolle una simulación basada en multimedia del juego de tejo. Use efectos de audio y visuales apropiados.

21.25 *(Juego de billar)* Cree una simulación basada en multimedia del juego de billar. Cada jugador toma turnos utilizando el ratón para posicionar un taco y pegarle a la bola en el ángulo apropiado para tratar de hacer que las demás bolas caigan en las buchacas. Su programa deberá llevar la puntuación.

21.26 *(Artista)* Diseñe un programa artístico en Java que proporcione a un artista una gran variedad de herramientas para dibujar, utilizar imágenes, animaciones, etcétera, para crear una muestra de arte dinámica con multimedia.

21.27 *(Diseñador de fuegos artificiales)* Cree un programa en Java que podría ser utilizado para crear una muestra de fuegos artificiales. Cree una variedad de demostraciones de fuegos artificiales. Después controle la activación de los fuegos artificiales para obtener un efecto máximo.

21.28 *(Diseñador de pisos)* Desarrolle un programa en Java que ayude a alguien a distribuir los muebles en su hogar. Agregue características que permitan a la persona lograr el mejor arreglo posible.

21.29 *(Crucigrama)* Los crucigramas son uno de los pasatiempos más populares. Desarrolle un programa de crucigramas basado en multimedia. Su programa debe permitir al jugador colocar y borrar palabras fácilmente. Enlace su programa con un diccionario computarizado extenso. Su programa deberá también tener la capacidad de sugerir palabras en las que se hayan llenado algunas letras. Proporcione otras características que faciliten el trabajo de los entusiastas de crucigramas.

21.30 *(Acertijo del 15)* Escriba un programa en Java basado en multimedia que permita al usuario jugar al 15. Este juego se lleva a cabo en un tablero de 4 por 4, para un total de 16 posiciones. Una de las posiciones está vacía. Las demás posiciones están ocupadas por 15 mosaicos, numerados del 1 al 15. Cualquier mosaico enseguida de la posición actual vacía puede moverse a esa posición, haciendo clic en el mosaico. Su programa deberá crear el tablero con los mosaicos desordenados. El objetivo es ordenar los mosaicos en forma secuencial, fila por fila.

21.31 *(Tiempo de reacción/Probador de precisión de reacción)* Cree un programa en Java que mueva una figura creada al azar alrededor de la pantalla. El usuario desplazará el ratón para atrapar y hacer clic en la figura. Puede variarse la velocidad y el tamaño de la figura. Su programa deberá llevar las estadísticas acerca de cuánto tiempo le lleva al usuario atrapar una figura de un tamaño dado. Probablemente al usuario le sea más difícil atrapar figuras que sean más pequeñas y se muevan más rápido.

21.32 *(Calendario/archivo de recordatorios)* Utilizando audio e imágenes, cree un calendario de propósito general y un archivo de "recordatorios". Por ejemplo, el programa deberá cantar "Feliz cumpleaños" cuando lo utilice en su cumpleaños. Haga que el programa muestre imágenes y reproduzca clips de audio asociados con eventos importantes. Además, haga que el programa le recuerde de antemano estos eventos importantes. Por ejemplo, sería bueno hacer que el programa le avisara con una semana de anticipación, para que pudiera recoger una tarjeta de felicitación apropiada para esa persona especial.

21.33 *(Imágenes giratorias)* Cree un programa en Java que le permita girar una imagen un cierto número de grados (hasta un máximo de 360 grados). El programa deberá permitirle a usted especificar que desea girar la imagen en forma continua. El programa deberá permitirle ajustar la velocidad de giro en forma dinámica.

21.34 *(Colorear imágenes y fotografías en blanco y negro)* Cree un programa en Java que le permita pintar de colores una fotografía en blanco y negro. Proporcione una paleta para seleccionar colores. Su programa deberá permitirle aplicar distintos colores en distintas regiones de la imagen.

21.35 *(Simulador Simpletron basado en multimedia)* Modifique el simulador Simpletron que desarrolló en los ejercicios de capítulos anteriores (ejercicios 7.34 a 7.36 y ejercicios 17.26 a 17.30), para incluir características de multimedia. Agregue sonidos tipo computadora para indicar que el Simpletron está ejecutando instrucciones. Agregue un sonido de vidrio quebrándose cuando ocurra un error fatal. Use luces intermitentes para indicar cuáles celdas de memoria o cuáles registros se están manipulando en ese momento. Use otras técnicas de multimedia según sea apropiado, para hacer que su simulador Simpletron sea más valioso para sus usuarios como una herramienta educativa.

22



Si un actor entra por la puerta, no tienes nada. Pero si entra por la ventana, tienes un problema.

—Billy Wilder

…la fuerza de los eventos despierta talentos adormilados.

—Edward Hoagland

Usted y yo veríamos la fotografía con más interés, si dejaran de preocuparse y aplicaran el sentido común al problema de registrar la apariencia visual de su propia era.

—Jessie Tarbox Beals

Componentes de la GUI: parte 2

OBJETIVOS

En este capítulo aprenderá a:

- Crear y manipular controles deslizables, menús, menús contextuales y ventanas.
- Cambiar la apariencia visual de una GUI, utilizando la apariencia visual adaptable (PLAF) de Swing.
- Crear una interfaz de múltiples documentos con `JDesktopPane` y `JInternalFrame`.
- Utilizar los administradores de esquemas adicionales.

Plan general

22.1 Introducción

En este capítulo continuaremos nuestro estudio acerca de las GUIs. Hablaremos sobre componentes y administradores de esquemas adicionales, y estableceremos las bases para crear las GUIs más complejas.

Empezaremos nuestra discusión con los menús, que permiten al usuario realizar con efectividad tareas en el programa. La apariencia visual de una GUI de Swing puede ser uniforme a través de todas las plataformas en las que se ejecuta un programa de Java, o la GUI puede personalizarse mediante el uso de la **apariencia visual adaptable (PLAF)** de Swing. Proporcionaremos un ejemplo que ilustra cómo cambiar entre la apariencia visual metálica de Swing (que tiene una apariencia y comportamiento similares entre las distintas plataformas), una apariencia visual que simula a **Motif** (una apariencia visual de UNIX muy popular) y una que simula la apariencia visual de Microsoft Windows.

Muchas de las aplicaciones de hoy en día utilizan una interfaz de múltiples documentos (MDI): una ventana principal (conocida también como la ventana padre) que contiene a otras ventanas (a menudo conocidas como ventanas hijas) para administrar varios documentos abiertos en paralelo. Por ejemplo, muchos programas de correo electrónico le permiten tener varias ventanas abiertas al mismo tiempo, para que pueda componer o leer varios mensajes de correo electrónico. En este capítulo demostraremos el uso de las clases de Swing para crear interfaces con múltiples documentos. Finalmente, terminaremos el capítulo con una serie de ejemplos acerca de los administradores de esquemas adicionales que están disponibles para organizar interfaces gráficas de usuario.

Swing es un tema extenso y complejo. Existen muchos más componentes de la GUI y herramientas de las que podemos presentar aquí. Presentaremos más componentes de la GUI de Swing en los capítulos restantes de este libro, a medida que se vayan necesitando. En nuestro libro *Advanced Java 2 Platform How to Program* hablamos sobre otros componentes y herramientas más avanzadas de Swing.

22.2 JSlider

Los objetos `JSlider` permiten al usuario seleccionar de entre un rango de valores enteros. La clase `JSlider` hereda de `JComponent`. En la figura 22.1 se muestra un objeto `JSlider` horizontal con **marcas** y el **indicador** que permite al usuario seleccionar un valor. Los objetos `JSlider` pueden personalizarse para mostrar marcas más distanciadas, marcas menos distanciadas y etiquetas para las marcas. También soportan el **ajuste a la marca**, en el que al colocar el indicador entre dos marcas, éste se ajusta a la marca más cercana.

La mayoría de los componentes de la GUI de Swing soportan las interacciones del usuario mediante el ratón y el teclado. Por ejemplo, si un objeto `JSlider` tiene el foco (es decir, que sea el componente de la GUI seleccio-

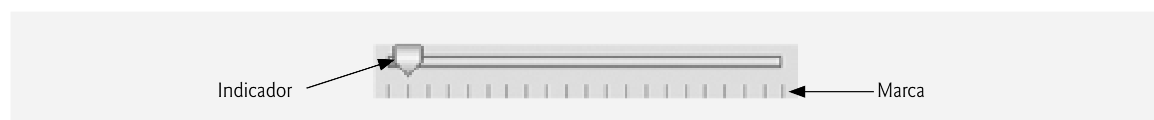


Figura 22.1 | Componente `JSlider` con orientación horizontal.

nado en ese momento, en la interfaz de usuario), la tecla de flecha izquierda y la tecla de flecha derecha hacen que el indicador del objeto `JSlider` se decremente o se incremente en 1, respectivamente. La tecla de flecha hacia abajo y la tecla de flecha hacia arriba también hacen que el indicador del objeto `JSlider` se decremente o incremente en 1 marca, respectivamente. Las teclas *Av Pág* (avance de página) y *Re Pág* (retroceso de página) hacen que el indicador del objeto `JSlider` se decremente o incremente en **incrementos de bloque** de una décima parte del rango de valores, respectivamente. La *tecla Inicio* desplaza el indicador hacia el valor mínimo del objeto `JSlider`, y la *tecla Fin* lo desplaza hacia el valor máximo del objeto `JSlider`.

Los objetos `JSlider` tienen una orientación horizontal o una vertical. Para un objeto `JSlider` horizontal, el valor mínimo se encuentra en el extremo izquierdo y el valor máximo, en el derecho. Para un objeto `JSlider` vertical, el valor mínimo se encuentra en el extremo inferior y el valor máximo, en el superior. Las posiciones de los valores mínimo y máximo del objeto `JSlider` se pueden invertir mediante la invocación del método **`setInverted`** de `JSlider` con el argumento `boolean true`. La posición relativa del indicador muestra el valor actual del objeto `JSlider`.

El programa de las figuras 22.2 y 22.4 permite al usuario ajustar el tamaño de un círculo dibujado en una subclase de `JPanel`, llamada `PanelOvalo` (figura 22.2). El usuario especifica el diámetro del círculo con un objeto `JSlider` horizontal. La clase `PanelOvalo` sabe cómo dibujar un círculo en sí misma, utilizando su propia variable de instancia `diametro` para determinar el diámetro del círculo; el `diametro` se utiliza como la anchura y altura del cuadro delimitador en el que se muestra el círculo. El valor del `diametro` se establece cuando el usuario interactúa con el objeto `JSlider`. El manejador de eventos llama al método `establecerDiametro` en la clase `PanelOvalo` para establecer el `diametro`, y llama a `repaint` para dibujar el nuevo círculo. La llamada a `repaint` resulta en una llamada al método `paintComponent` de `PanelOvalo`.

```java
// Fig. 22.2: PanelOvalo.java
// Una clase JPanel personalizada.
import java.awt.Graphics;
import java.awt.Dimension;
import javax.swing.JPanel;

public class PanelOvalo extends JPanel
{
   private int diametro = 10; // diámetro predeterminado de 10

   // dibuja un óvalo del diámetro especificado
   public void paintComponent( Graphics g )
   {
      super.paintComponent( g );

      g.fillOval( 10, 10, diametro, diametro ); // dibuja un círculo
   } // fin del método paintComponent

   // valida y establece el diámetro, después vuelve a pintar
   public void establecerDiametro( int nuevoDiametro )
   {
      // si el diámetro es inválido, usa el valor predeterminado de 10
      diametro = ( nuevoDiametro >= 0 ? nuevoDiametro : 10 );
      repaint(); // vuelve a pintar el panel
   } // fin del método establecerDiametro

   // lo utiliza el administrador de esquemas para determinar el tamaño preferido
   public Dimension getPreferredSize()
   {
      return new Dimension( 200, 200 );
   } // fin del método getPreferredSize

   // lo utiliza el administrador de esquemas para determinar el tamaño mínimo
```

Figura 22.2 | Subclase de `JPanel` para dibujar círculos de un diámetro especificado. (Parte 1 de 2).

```
34      public Dimension getMinimumSize()
35      {
36         return getPreferredSize();
37      } // fin del método getMinimumSize
38   } // fin de la clase PanelOvalo
```

Figura 22.2 | Subclase de `JPanel` para dibujar círculos de un diámetro especificado. (Parte 2 de 2).

La clase `PanelOvalo` (figura 22.2) contiene un método `paintComponent` (líneas 12 a 17) que dibuja un óvalo relleno (un círculo en este ejemplo), un método `establecerDiametro` (líneas 20 a 25) que modifica el `diametro` del círculo y vuelve a pintar (`repaint`) el `PanelOvalo`, un método **`getPreferredSize`** (líneas 28 a 31) que devuelve la anchura y altura preferidas de un objeto `PanelOvalo`, y un método **`getMinimumSize`** (líneas 34 a 37) que devuelve la anchura y altura mínimas de un objeto `PanelOvalo`.



Observación de apariencia visual 22.1

Si un nuevo componente de la GUI tiene una anchura y altura mínimas (es decir, que unas medidas menores hagan que el componente se muestre incorrectamente en la pantalla), debe sobrescribir el método `getMinimumSize` *para devolver la anchura y altura mínimas como una instancia de la clase* `Dimension`*.*



Observación de ingeniería de software 22.1

Para muchos componentes de la GUI, el método `getMinimumSize` *se implementa para devolver el resultado de una llamada al método* `getPreferredSize` *de ese componente.*

La clase `MarcoSlider` (figura 22.3) crea el objeto `JSlider` que controla el diámetro del círculo. El constructor de la clase `MarcoSlider` (líneas 17 a 45) crea el objeto `PanelOvalo` llamado `miPanel` (línea 21) y establece su color de fondo (línea 22). En las líneas 25 y 26 se crea el objeto `JSlider` llamado `diametroJSlider` para controlar el diámetro del círculo que se dibuja en el `PanelOvalo`. El constructor de `JSlider` recibe cuatro argumentos. El primero especifica la orientación del `diametroJSlider`, que es `HORIZONTAL` (una constante en la interfaz `SwingConstants`). Los argumentos segundo y tercero indican los valores enteros mínimo y máximo en el rango de valores para este objeto `JSlider`. El último argumento indica que el valor inicial del objeto `JSlider` (es decir, en dónde se muestra el indicador) debe ser `10`.

En las líneas 27 y 28 se personaliza la apariencia del objeto `JSlider`. El método **`setMajorTickSpacing`** indica que cada marca principal representa 10 valores en el rango soportado por el objeto `JSlider`. El método **`setPaintTicks`** con un argumento `true` indica que las marcas deben mostrarse (no se muestran de manera predeterminada). Para los otros métodos que se utilizan para personalizar la apariencia de un objeto **`JSlider`**, consulte la documentación en línea de la clase `JSlider` (`java.sun.com/javase/6/docs/api/javax/swing/JSlider.html`).

```
1    // Fig. 22.3: MarcoSlider.java
2    // Uso de objetos JSlider para cambiar el tamaño de un óvalo.
3    import java.awt.BorderLayout;
4    import java.awt.Color;
5    import javax.swing.JFrame;
6    import javax.swing.JSlider;
7    import javax.swing.SwingConstants;
8    import javax.swing.event.ChangeListener;
9    import javax.swing.event.ChangeEvent;
10
11   public class MarcoSlider extends JFrame
12   {
13      private JSlider diametroJSlider; // control deslizable para seleccionar el diámetro
```

Figura 22.3 | Valor de `JSlider` que se utiliza para determinar el diámetro de un círculo. (Parte 1 de 2).

```
14     private PanelOvalo miPanel; // panel para dibujar el círculo
15
16     // constructor sin argumentos
17     public MarcoSlider()
18     {
19        super( "Demostracion de JSlider" );
20
21        miPanel = new PanelOvalo(); // crea panel para dibujar el círculo
22        miPanel.setBackground( Color.YELLOW ); // establece el color de fondo en amarillo
23
24        // establece objeto JSlider para controlar el valor del diámetro
25        diametroJSlider =
26           new JSlider( SwingConstants.HORIZONTAL, 0, 200, 10 );
27        diametroJSlider.setMajorTickSpacing( 10 ); // crea una marca cada 10
28        diametroJSlider.setPaintTicks( true ); // dibuja las marcas en el control
          deslizable
29
30        // registra el componente que escucha los eventos del objeto JSlider
31        diametroJSlider.addChangeListener(
32
33           new ChangeListener() // clase interna anónima
34           {
35              // maneja el cambio en el valor del control deslizable
36              public void stateChanged( ChangeEvent e )
37              {
38                 miPanel.establecerDiametro( diametroJSlider.getValue() );
39              } // fin del método stateChanged
40           } // fin de la clase interna anónima
41        ); // fin de la llamada a addChangeListener
42
43        add( diametroJSlider, BorderLayout.SOUTH ); // agrega el control deslizable al marco
44        add( miPanel, BorderLayout.CENTER ); // agrega el panel al marco
45     } // fin del constructor de MarcoSlider
46  } // fin de la clase MarcoSlider
```

Figura 22.3 | Valor de `JSlider` que se utiliza para determinar el diámetro de un círculo. (Parte 2 de 2).

Los objetos `JSlider` generan eventos **`ChangeEvent`** (paquete `javax.swing.event`) en respuesta a las interacciones del usuario. Un objeto de una clase que implementa a la interfaz **`ChangeListener`** (paquete `javax.swing.event`) y declara el método **`stateChanged`** puede responder a los eventos `ChangeEvent`. En las líneas 31 a 41 se registra un objeto `ChangeListener` para manejar los eventos de `diametroJSlider`. Cuando se llama al método `stateChanged` (líneas 36 a 39) en respuesta a una interacción del usuario, en la línea 38 se hace una llamada al método `establecerDiametro` de `miPanel` y se pasa el valor actual del objeto `JSlider` como argumento. El método **`getValue`** de `JSlider` devuelve la posición actual del indicador.

```
// Fig. 22.4: DemoSlider.java
// Prueba de MarcoSlider.
import javax.swing.JFrame;

public class DemoSlider
{
   public static void main( String args[] )
   {
      MarcoSlider marcoSlider = new MarcoSlider();
      marcoSlider.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
```

Figura 22.4 | Clase de prueba para `MarcoSlider`. (Parte 1 de 2).

```
      marcoSlider.setSize( 220, 270 ); // establece el tamaño del marco
      marcoSlider.setVisible( true ); // muestra el marco
   } // fin de main
} // fin de la clase DemoSlider
```

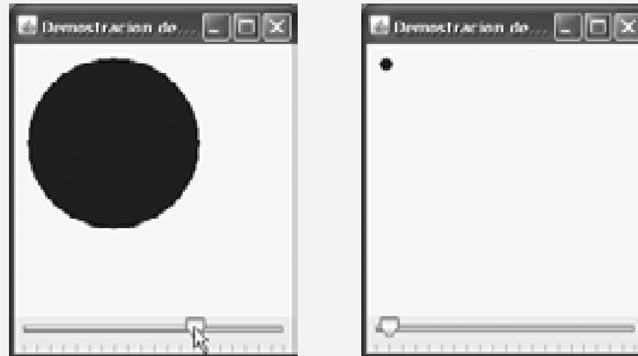


Figura 22.4 | Clase de prueba para `MarcoSlider`. (Parte 2 de 2).

22.3 Ventanas: observaciones adicionales

En esta sección, hablaremos sobre algunas cuestiones importantes de `JFrame`. Un objeto `JFrame` es una **ventana** con una **barra de título** y un **borde**. La clase `JFrame` es una subclase de `java.awt.Frame` (que es una subclase de `java.awt.Window`). Como tal, `JFrame` es uno de los pocos componentes de la GUI de Swing que no es un componente de la GUI ligero. Cuando se muestra una ventana desde un programa de Java, ésta se proporciona mediante el kit de herramientas de ventana de la plataforma local y, por lo tanto, la ventana tendrá la misma apariencia que las otras ventanas que se muestren en esa plataforma. Cuando se ejecute una aplicación de Java en una Macintosh en donde se muestre una ventana, la barra de título y los bordes de la ventana tendrán la misma apariencia que las ventanas de las demás aplicaciones Macintosh. Cuando se ejecute una aplicación de Java en Microsoft Windows y se muestre una ventana, la barra de título y los bordes tendrán la misma apariencia que las ventanas de las demás aplicaciones Microsoft Windows. Y cuando se ejecute una aplicación de Java en una plataforma UNIX y se muestre una ventana, la barra de título y los bordes de la ventana tendrán la misma apariencia que las ventanas de las demás aplicaciones UNIX en esa plataforma.

La clase `JFrame` soporta tres operaciones cuando el usuario cierra la ventana. De manera predeterminada, una ventana se oculta (es decir, se quita de la pantalla) cuando el usuario la cierra. Esto puede controlarse con el método **`setDefaultCloseOperation`** de `JFrame`. La interfaz **`WindowConstants`** (paquete `javax.swing`), que es implementada por la clase `JFrame`, declara tres constantes para ser utilizadas con este método: `DISPOSE_ON_CLOSE`, `DO_NOTHING_ON_CLOSE` y `HIDE_ON_CLOSE` (el valor predeterminado). Algunas plataformas sólo permiten que se muestre un número limitado de ventanas en la pantalla. Por lo tanto, una ventana es un valioso recurso que debe regresarse al sistema cuando ya no se necesite. La clase `Window` (una superclase indirecta de `JFrame`) declara el método **`dispose`** para este fin. Cuando ya no se necesita un objeto `Window` en una aplicación, se debe desechar explícitamente este objeto `Window`. Esto puede hacerse llamando al método `dispose` de `Window`, o llamando al método `setDefaultCloseOperation` con el argumento `WindowConstants.DISPOSE_ON_CLOSE`. Al terminar una aplicación se devuelven todos los recursos de ventana al sistema. Al establecer la operación de cierre predeterminada en `DO_NOTHING_ON_CLOSE`, el programa determinará lo que debe hacer cuando el usuario le indique que debe cerrar la ventana.



Tip de rendimiento 22.1

Una ventana es un recurso imprescindible del sistema. Devuélvala al sistema cuando ya no sea necesaria.

De manera predeterminada, una ventana no se muestra en la pantalla sino hasta que el programa invoque al método `setVisible` de esta ventana (heredado de la clase `java.awt.Component`) con un argumento `true`. Ade-

más, debe establecerse el tamaño de una ventana mediante una llamada al método `setSize` (heredado de la clase `java.awt.Component`). La posición que tendrá la ventana cuando aparezca en la pantalla se especifica mediante el método **`setLocation`** (heredado de la clase `java.awt.Component`).



Error común de programación 22.1

Olvidar llamar al método `setVisible` en una ventana es un error lógico en tiempo de ejecución; la ventana no se mostrará en pantalla.



Error común de programación 22.2

Olvidar llamar al método `setSize` en una ventana es un error lógico en tiempo de ejecución; sólo aparecerá la barra de título.

Las ventanas generan **eventos de ventana** cuando el usuario las manipula. Los componentes de escucha de eventos se registran para eventos de ventana mediante el método **`addWindowListener`** de la clase `Window`. La interfaz **`WindowListener`** proporciona siete métodos manejadores de eventos de ventana: **`windowActivated`** (se invoca cuando el usuario hace que la ventana sea la ventana activa), **`windowClosed`** (se invoca cuando se cierra la ventana), **`windowClosing`** (se invoca cuando el usuario inicia el cierre de la ventana), **`windowDeactivated`** (se invoca cuando el usuario hace que otra ventana sea la ventana activa), **`windowDeiconified`** (se invoca cuando el usuario restaura una ventana, después de que ha sido minimizada), **`windowIconified`** (se invoca cuando el usuario minimiza una ventana) y **`windowOpened`** (se invoca cuando un programa muestra por primera vez una ventana en la pantalla).

22.4 Uso de menús con marcos

Los **menús** son una parte integral de las GUIs. Permiten al usuario realizar acciones sin saturar innecesariamente una GUI con componentes adicionales. En las GUIs de Swing, los menús pueden adjuntarse solamente a los objetos de clases que proporcionen el método **`setJMenuBar`**. Dos de esas clases son `JFrame` y `JApplet`. Las clases utilizadas para declarar menús son `JMenuBar`, `JMenu`, `JMenuItem`, `JCheckBoxMenuItem` y la clase `JRadioButtonMenuItem`.



Observación de apariencia visual 22.2

Los menús simplifican las GUIs, ya que se pueden ocultar componentes dentro de ellos. Estos componentes estarán visibles sólo cuando el usuario los busque, al seleccionar el menú.

La clase **`JMenuBar`** (una subclase de `JComponent`) contiene los métodos necesarios para administrar una **barra de menús**, la cual es un contenedor para los menús. La clase **`JMenu`** (una subclase de `javax.swing.JMenuItem`) contiene los métodos necesarios para administrar menús. Los menús contienen elementos de menú, y pueden agregarse a barras de menús o a otros menús, como submenús. Cuando se hace clic en un menú, éste se expande para mostrar su lista de elementos de menú.

La clase **`JMenuItem`** (una subclase de **`javax.swing.AbstractButton`**) contiene los métodos necesarios para administrar **elementos de menú**. Un elemento de menú es un componente de la GUI que está dentro de un menú y que, cuando se selecciona, produce un evento de acción. Un elemento de menú puede usarse para iniciar una acción, o puede ser un **submenú** que proporcione más elementos de menú que el usuario pueda seleccionar. Los submenús son útiles para agrupar en un menú los elementos de menú que estén relacionados.

La clase **`JCheckBoxMenuItem`** (una subclase de **`javax.swing.JMenuItem`**) contiene los métodos necesarios para administrar los elementos de menú que pueden activarse o desactivarse. Cuando se selecciona un objeto `JCheckBoxMenuItem`, aparece una marca de verificación a la izquierda del elemento de menú. Cuando se selecciona de nuevo el mismo objeto `JCheckBoxMenuItem`, se quita la marca de verificación a la izquierda del elemento de menú.

La clase **`JRadioButtonMenuItem`** (una subclase de `javax.swing.JMenuItem`) contiene los métodos necesarios para administrar elementos de menú que pueden activarse o desactivarse, de manera parecida a los objetos `JCheckBoxMenuItem`. Cuando se mantienen varios objetos `JRadioButtonMenuItem` como parte de un objeto `ButtonGroup`, sólo puede seleccionarse un elemento en el grupo a la vez. Cuando se selecciona un objeto

`JRadioButtonMenuItem`, aparece un círculo relleno a la izquierda del elemento de menú. Cuando se selecciona otro objeto `JCheckBoxMenuItem`, se quita el círculo relleno a la izquierda del elemento de menú previamente seleccionado.

La aplicación de las figuras 22.5 y 22.6 demuestra el uso de varios elementos de menú y cómo especificar caracteres especiales (llamados **nemónicos**) que pueden proporcionar un acceso rápido a un menú o elemento de menú, desde el teclado. Los nemónicos pueden usarse con todas las subclases de `javax.swing.AbstractButton`.

La clase `MarcoMenu` (figura 22.5) declara los componentes de la GUI y el manejo de eventos para los elementos de menú. La mayor parte del código en esta aplicación aparece en el constructor de la clase (líneas 34 a 151).

```
1  // Fig. 22.5: MarcoMenu.java
2  // Demostración de los menús.
3  import java.awt.Color;
4  import java.awt.Font;
5  import java.awt.BorderLayout;
6  import java.awt.event.ActionListener;
7  import java.awt.event.ActionEvent;
8  import java.awt.event.ItemListener;
9  import java.awt.event.ItemEvent;
10 import javax.swing.JFrame;
11 import javax.swing.JRadioButtonMenuItem;
12 import javax.swing.JCheckBoxMenuItem;
13 import javax.swing.JOptionPane;
14 import javax.swing.JLabel;
15 import javax.swing.SwingConstants;
16 import javax.swing.ButtonGroup;
17 import javax.swing.JMenu;
18 import javax.swing.JMenuItem;
19 import javax.swing.JMenuBar;
20
21 public class MarcoMenu extends JFrame
22 {
23    private final Color valoresColores[] =
24       { Color.BLACK, Color.BLUE, Color.RED, Color.GREEN };
25    private JRadioButtonMenuItem elementosColores[]; // elementos del menú colores
26    private JRadioButtonMenuItem fuentes[]; // elementos del menú fuentes
27    private JCheckBoxMenuItem elementosEstilos[]; // elementos del menú estilos
28    private JLabel mostrarJLabel; // muestra texto de ejemplo
29    private ButtonGroup fuentesButtonGroup; // administra elementos del menú fuentes
30    private ButtonGroup coloresButtonGroup; // administra elementos del menú colores
31    private int estilo; // se utiliza para crear el estilo para la fuente
32
33    // el constructor sin argumentos establece la GUI
34    public MarcoMenu()
35    {
36       super( "Uso de objetos JMenu" );
37
38       JMenu menuArchivo = new JMenu( "Archivo" ); // crea el menú archivo
39       menuArchivo.setMnemonic( 'A' ); // establece el nemónico a A
40
41       // crea el elemento de menú Acerca de...
42       JMenuItem elementoAcercaDe = new JMenuItem( "Acerca de..." );
43       elementoAcercaDe.setMnemonic( 'c' ); // establece el nemónico a c
44       menuArchivo.add( elementoAcercaDe ); // agrega el elemento elementoAcercaDe al
         menú archivo
45       elementoAcercaDe.addActionListener(
46
```

Figura 22.5 | Objetos `JMenu` y nemónicos. (Parte 1 de 4).

```
         new ActionListener() // clase interna anónima
         {
            // muestra cuadro de diálogo de mensaje cuando el usuario selecciona Acerca de...
            public void actionPerformed( ActionEvent evento )
            {
               JOptionPane.showMessageDialog( MarcoMenu.this,
                  "Este es un ejemplo\ndel uso de menus",
                  "Acerca de", JOptionPane.PLAIN_MESSAGE );
            } // fin del método actionPerformed
         } // fin de la clase interna anónima
      ); // fin de la llamada a addActionListener

      JMenuItem elementoSalir = new JMenuItem( "Salir" ); // crea el elemento salir
      elementoSalir.setMnemonic( 'S' ); // establece el nemónico a S
      menuArchivo.add( elementoSalir ); // agrega elemento salir al menú archivo
      elementoSalir.addActionListener(

         new ActionListener() // clase interna anónima
         {
            // termina la aplicación cuando el usuario hace clic en elementoSalir
            public void actionPerformed( ActionEvent evento )
            {
               System.exit( 0 ); // sale de la aplicación
            } // fin del método actionPerformed
         } // fin de la clase interna anónima
      ); // fin de la llamada a addActionListener

      JMenuBar barra = new JMenuBar(); // crea la barra de menús
      setJMenuBar( barra ); // agrega la barra de menús a la aplicación
      barra.add( menuArchivo ); // agrega el menú archivo a la barra de menús

      JMenu menuFormato = new JMenu( "Formato" ); // crea el menú formato
      menuFormato.setMnemonic( 'F' ); // establece el nemónico a F

      // arreglo que enlista la cadena colores
      String colores[] = { "Negro", "Azul", "Rojo", "Verde" };

      JMenu menuColor = new JMenu( "Color" ); // crea el menú color
      menuColor.setMnemonic( 'C' ); // establece el nemónico a C

      // crea los elementos de menú de los botones de opción para los colores
      elementosColores = new JRadioButtonMenuItem[ colores.length ];
      coloresButtonGroup = new ButtonGroup(); // administra los colores
      ManejadorElementos manejadorElementos = new ManejadorElementos(); // manejador para colores

      // crea los elementos de menú del botón de opción color
      for ( int cuenta = 0; cuenta < colores.length; cuenta++ )
      {
         elementosColores[ cuenta ] =
            new JRadioButtonMenuItem( colores[ cuenta ] ); // crea elemento
         menuColor.add( elementosColores[ cuenta ] ); // agrega elemento al menú color
         coloresButtonGroup.add( elementosColores[ cuenta ] ); // lo agrega al grupo
         elementosColores[ cuenta ].addActionListener( manejadorElementos );
      } // fin de for

      elementosColores[ 0 ].setSelected( true ); // selecciona el primer elemento de Color

```

Figura 22.5 | Objetos `JMenu` y nemónicos. (Parte 2 de 4).

```
      menuFormato.add( menuColor ); // agrega el menú color al menú formato
      menuFormato.addSeparator(); // agrega un separador en el menú

      // arreglo que enlista los nombres de las fuentes
      String nombresFuentes[] = { "Serif", "Monospaced", "SansSerif" };
      JMenu menuFuente = new JMenu( "Fuente" ); // crea el menú fuente
      menuFuente.setMnemonic( 'u' ); // establece el nemónico a u

      // crea elementos de menú de botones de opción para los nombres de las fuentes
      fuentes = new JRadioButtonMenuItem[ nombresFuentes.length ];
      fuentesButtonGroup = new ButtonGroup(); // administra los nombres de las fuentes

      // crea elementos de menú de botones de opción de Fuente
      for ( int cuenta = 0; cuenta < fuentes.length; cuenta++ )
      {
         fuentes[ cuenta ] = new JRadioButtonMenuItem( nombresFuentes[ cuenta ] );
         menuFuente.add( fuentes[ cuenta ] ); // agrega fuente al menú fuente
         fuentesButtonGroup.add( fuentes[ cuenta ] ); // agrega al grupo de botones
         fuentes[ cuenta ].addActionListener( manejadorElementos ); // agrega el manejador
      } // fin de for

      fuentes[ 0 ].setSelected( true ); // selecciona el primer elemento del menú Fuente
      menuFuente.addSeparator(); // agrega barra separadora al menú fuente

      String nombresEstilos[] = { "Negrita", "Cursiva" }; // nombres de los estilos
      elementosEstilos = new JCheckBoxMenuItem[ nombresEstilos.length ];
      ManejadorEstilos manejadorEstilos = new ManejadorEstilos(); // manejador de estilos

      // crea elementos de menú de la casilla de verificación de estilo
      for ( int cuenta = 0; cuenta < nombresEstilos.length; cuenta++ )
      {
         elementosEstilos[ cuenta ] =
            new JCheckBoxMenuItem( nombresEstilos[ cuenta ] ); // para el estilo
         menuFuente.add( elementosEstilos[ cuenta ] ); // agrega al menú fuente
         elementosEstilos[ cuenta ].addItemListener( manejadorEstilos ); // manejador
      } // fin de for

      menuFormato.add( menuFuente ); // agrega el menú Fuente al menú Formato
      barra.add( menuFormato ); // agrega el menú Formato a la barra de menús

      // establece la etiqueta para mostrar el texto
      mostrarJLabel = new JLabel( "Texto de ejemplo", SwingConstants.CENTER );
      mostrarJLabel.setForeground( valoresColores[ 0 ] );
      mostrarJLabel.setFont( new Font( "'Serif", Font.PLAIN, 72 ) );

      getContentPane().setBackground( Color.CYAN ); // establece el color de fondo
      add( mostrarJLabel, BorderLayout.CENTER ); // agrega mostrarJLabel
   } // fin del constructor de MarcoMenu

   // clase interna para manejar los eventos de acción de los elementos de menú
   private class ManejadorElementos implements ActionListener
   {
      // procesa las selecciones de color y fuente
      public void actionPerformed( ActionEvent evento )
      {
         // procesa la selección del color
```

Figura 22.5 | Objetos `JMenu` y nemónicos. (Parte 3 de 4).

```
         for ( int cuenta = 0; cuenta < elementosColores.length; cuenta++ )
         {
            if ( elementosColores[ cuenta ].isSelected() )
            {
               mostrarJLabel.setForeground( valoresColores[ cuenta ] );
               break;
            } // fin de if
         } // fin de for

         // procesa la selección de fuente
         for ( int cuenta = 0; cuenta < fuentes.length; cuenta++ )
         {
            if ( evento.getSource() == fuentes[ cuenta ] )
               {
                  mostrarJLabel.setFont(
                     new Font( fuentes[ cuenta ].getText(), estilo, 72 ) );
               } // fin de if
            } // fin de for

            repaint(); // vuelve a dibujar la aplicación
         } // fin del método actionPerformed
      } // fin de la clase ManejadorElementos

      // clase interna para manejar los eventos de los elementos de menú de las
         casillas de verificación
      private class ManejadorEstilos implements ItemListener
      {
         // procesa las selecciones de estilo de las fuentes
         public void itemStateChanged( ItemEvent e )
         {
            estilo = 0; // inicializa el estilo

         // comprueba la selección de negrita
         if ( elementosEstilos[ 0 ].isSelected() )
            estilo += Font.BOLD; // agrega negrita al estilo

         // comprueba la selección de cursiva
         if ( elementosEstilos[ 1 ].isSelected() )
            estilo += Font.ITALIC; // agrega cursiva al estilo

         mostrarJLabel.setFont(
            new Font( mostrarJLabel.getFont().getName(), estilo, 72 ) );
         repaint(); // vuelve a dibujar la aplicación
      } // fin del método itemStateChanged
   } // fin de la clase ManejadorEstilos
} // fin de la clase MarcoMenu
```

Figura 22.5 | Objetos `JMenu` y nemónicos. (Parte 4 de 4).

```
// Fig. 22.6: PruebaMenu.java
// Prueba de MarcoMenu.
import javax.swing.JFrame;

public class PruebaMenu
{
   public static void main( String args[] )
   {
      MarcoMenu marcoMenu = new MarcoMenu(); // crea objeto MarcoMenu
```

Figura 22.6 | Clase de prueba para `MarcoMenu`. (Parte 1 de 2).

```
        marcoMenu.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
        marcoMenu.setSize( 600, 200 ); // establece el tamaño del marco
        marcoMenu.setVisible( true ); // muestra el marco
    } // fin de main
} // fin de la clase PruebaMenu
```



Figura 22.6 | Clase de prueba para `MarcoMenu`. (Parte 2 de 2).

En las líneas 38 a 76 se establece el menú **Archivo** y se adjunta a la barra de menús. El menú **Archivo** contiene un elemento de menú **Acerca de...**, el cual muestra un cuadro de diálogo de mensaje al ser seleccionado, y un elemento de menú **Salir** que puede seleccionarse para terminar la aplicación.

En la línea 38 se crea el objeto `JMenu` y se pasa a su constructor la cadena `"Archivo"` como el nombre del menú. En la línea 39 se utiliza el método **`setMnemonic`** (heredado de la clase `AbstractButton`) para indicar que `A` es el nemónico para este menú. Al oprimir las teclas *Alt* y *A* se abre el menú, así como cuando se hace clic en el menú con el ratón. En la GUI, el carácter nemónico en el nombre del menú se muestra subrayado. (Vea las capturas de pantalla de la figura 22.6.)



Observación de apariencia visual 22.3

Los nemónicos proporcionan un acceso rápido a los comandos de menú y de botón, por medio del teclado.



Observación de apariencia visual 22.4

Deben utilizarse distintos nemónicos para cada uno de los botones o elementos de menú. Generalmente, la primera letra en la etiqueta del elemento de menú o del botón es la que se utiliza como nemónico. Si varios botones o elementos de menú inician con la misma letra, seleccione la siguiente letra más prominente en el nombre (por ejemplo `u` *se selecciona comúnmente para el elemento de menú* `Guardar como`*, del menú* `Archivo`*).*

En las líneas 42 y 43 se crea el objeto `JMenuItem elementoAcercaDe` con el texto `"Acerca de..."` y se establece su nemónico en la letra `c`. (No se utiliza `A` porque esa letra es el nemónico del menú **Archivo**). Este elemento de menú se agrega al objeto `menuArchivo` en la línea 44 mediante el método **`add`** de `JMenu`. Para tener acceso al elemento **Acerca de...** a través del teclado, oprima la tecla *Alt* y la letra `A` para abrir el menú **Archivo**, después oprima `c` para seleccionar el elemento de menú **Acerca de...**. En las líneas 47 a 56 se crea un objeto `ActionListener` para procesar el evento de acción del objeto `elementoAcercaDe`. En las líneas 52 a 54 se muestra un cuadro de diálogo de mensaje. En la mayoría de los usos anteriores de `showMessageDialog`, el primer argumento ha sido `null`. El propósito del primer argumento es especificar la **ventana padre** para el cuadro de diálogo, la cual ayuda a determinar en dónde se mostrará el cuadro de diálogo. Si la ventana padre se especifica como `null`, el cuadro de diálogo aparece en el centro de la pantalla. En caso contrario, el cuadro de diálogo aparece centrado sobre la

ventana padre especificada. En este ejemplo, el programa especifica la ventana padre con `PruebaMenu.this`; la referencia `this` del objeto `PruebaMenu`. Al utilizar la referencia `this` en una clase interna, si se especifica la palabra `this` por sí sola, se hace referencia al objeto de la clase interna. Para hacer referencia a la referencia `this` del objeto de la clase externa, debe calificar a `this` con el nombre de la clase externa y un punto (`.`).

Los cuadros de diálogo generalmente son modales. Un **cuadro de diálogo modal** no permite el acceso a ninguna de las otras ventanas en la aplicación, sino hasta que se cierre. Los cuadros de diálogo que se muestran con la clase `JOptionPane` son cuadros de diálogo modales. La clase **`JDialog`** puede usarse para crear sus propios cuadros de diálogo modales o no modales.

En las líneas 59 a 72 se crea el elemento de menú `elementoSalir`, se establece su nemónico en `S`, se agrega a `menuArchivo` y se registra un objeto `ActionListener` que termina la aplicación cuando el usuario selecciona `elementoSalir`.

En las líneas 74 a 76 se crea el objeto `JMenuBar`, se adjunta a la ventana de aplicación mediante el método `setJMenuBar` de `JFrame` y se utiliza el método **`add`** de `JMenuBar` para adjuntar el `menuArchivo` a la barra de menús `JMenuBar`.



Error común de programación 22.3

Si olvida establecer la barra de menús mediante el método `setJMenuBar` de `JFrame`, la barra de menús no se mostrará en el objeto `JFrame`.



Observación de apariencia visual 22.5

Los menús generalmente aparecen de izquierda a derecha, en el orden en el que se agregan a un objeto `JMenuBar`.

En las líneas 78 a 79 se crea el menú llamado `menuFormato` y se establece su nemónico en `F`.

En las líneas 84 y 85 se crea el menú llamado `menuColor` (éste será un submenú en el menú **Formato**) y se establece su nemónico en `C`. En la línea 88 se crea el arreglo `JRadioButtonMenuItem` llamado `elementosColores`, el cual hace referencia a los elementos de menú de `menuColor`. En la línea 89 se crea el objeto `ButtonGroup` llamado `grupoColores`, el cual asegura que sólo se seleccione uno de los elementos de menú del submenú **Color** en un momento dado. En la línea 90 se crea una instancia de la clase interna `ManejadorElementos` (declarada en las líneas 154 a 181) para responder a las selecciones del submenú **Color** y del submenú **Fuente** (que describiremos en breve). En la instrucción `for` de las líneas 93 a 100 se crea a cada uno de los objetos `JRadioButtonMenuItem` en el arreglo `elementosColor`, se agrega cada uno de los elementos de menú a `menuColor` y a `grupoColores`, y se registra el objeto `ActionListener` para cada elemento de menú.

En la línea 102 se utiliza el método **`setSelected`** de `AbstractButton` para seleccionar el primer elemento en el arreglo `elementosColor`. En la línea 104 se agrega `menuColor` como un submenú de `menuFormato`. En la línea 105 se invoca al método **`addSeparator`** de `JMenu` para agregar una línea **separadora** horizontal al menú.



Observación de apariencia visual 22.6

Un submenú se crea agregando a un menú como elemento de otro menú. Cuando el ratón se coloca sobre un submenú (o cuando se oprime el nemónico de ese submenú), éste se expande para mostrar sus elementos de menú.



Observación de apariencia visual 22.7

Pueden agregarse separadores a un menú para agrupar los elementos de menú en forma lógica.



Observación de apariencia visual 22.8

Puede agregarse cualquier componente de la GUI ligero (es decir, un componente de cualquier subclase de la clase `JComponent`) a un objeto `JMenu` o `JMenuBar`.

En las líneas 108 a 126 se crean el submenú **Fuente** y varios objetos `JRadioButtonMenuItem`, y se selecciona el primer elemento del arreglo `JRadioButtonMenuItem` llamado `fuentes`. En la línea 129 se crea un arreglo `JCheckBoxMenuItem` para representar a los elementos de menú que especifican los estilos negrita y cursiva para las fuentes. En la línea 130 se crea una instancia de la clase interna `ManejadorEstilos` (declarada en las líneas 184 a 203) para responder a los eventos de `JCheckBoxMenuItem`. La instrucción `for` de las líneas 133 a 139 crea a cada objeto `JCheckBoxMenuItem`, agrega cada uno de los elementos de menú a `menuFuente` y registra el objeto

`ItemListener` para cada elemento de menú. En la línea 141 se agrega `menuFuente` como un submenú de `menuFormato`. En la línea 142 se agrega el `menuFormato` a la `barra` (de menús).

En las líneas 145 a 147 se crea un objeto `JLabel` para el cual los elementos del menú **Formato** controlan el tipo de letra, su color y estilo. El color inicial de primer plano se establece con el primer elemento del arreglo `valoresColor` (`Color.BLACK`) mediante la invocación al método **`setForeground`** de `JComponent`, y el tipo de letra inicial se establece en `Serif` con estilo `PLAIN` y tamaño de `72` puntos. En la línea 149 se establece el color de fondo del panel de contenido de la ventana en cyan, y en la línea 150 se adjunta el objeto `JLabel` a la región `CENTER` del esquema `BorderLayout` del panel de contenido.

El método `actionPerformed` de `ManejadorElementos` (líneas 157 a 180) utiliza dos instrucciones `for` para determinar cuál fue el elemento de menú de fuente o de color que generó el evento, y establece la fuente o el color del objeto `JLabel` llamado `mostrarEtiqueta`, respectivamente. La condición `if` de la línea 162 utiliza el método **`isSelected`** de `AbstractButton` para determinar cuál fue el objeto `JRadioButtonMenuItem` seleccionado. La condición `if` de la línea 172 invoca al método `getSource` del objeto evento para obtener una referencia al objeto `JRadioButtonMenuItem` que generó el evento. En la línea 175 se utiliza el método `getText` de `AbstractButton` para obtener el nombre del tipo de letra del elemento de menú.

El programa llama al método `itemStateChanged` de `ManejadorEstilo` (líneas 187 a 202) si el usuario selecciona un objeto `JCheckBoxMenuItem` en el `menuFuente`. En las líneas 192 y 196 se determina si uno o ambos objetos `JCheckBoxMenuItem` se seleccionan, y se utiliza su estado combinado para determinar el nuevo estilo de fuente.

22.5 JPopupMenu

Muchas de las aplicaciones de computadora de la actualidad proporcionan lo que se conoce como **menús contextuales sensibles al contexto**. En Swing, tales menús se crean con la clase **`JPopupMenu`** (una subclase de `JComponent`). Estos menús proporcionan opciones específicas al componente para el cual se generó el **evento de desencadenamiento del menú contextual**. En la mayoría de los sistemas, este evento de desencadenamiento ocurre cuando el usuario oprime y suelta el botón derecho del ratón.



Observación de apariencia visual 22.9

El evento de desencadenamiento del menú contextual es específico para cada plataforma. En la mayoría de las plataformas que utilizan un ratón con varios botones, el evento de desencadenamiento del menú contextual ocurre cuando el usuario hace clic con el botón derecho del ratón en un componente que tiene soporte para un menú contextual.

La aplicación de las figuras 22.7 a 22.8 crea un objeto `JPopupMenu`, el cual permite al usuario seleccionar uno de tres colores y cambiar el color de fondo de la ventana. Cuando el usuario hace clic con el botón derecho del ratón en el fondo de la ventana `PruebaContextual`, aparece un objeto `JPopupMenu`, el cual contiene unos colores. Si el usuario hace clic en un objeto `JRadioButtonMenuItem` para seleccionar un color, el método `actionPerformed` de `ManejadorElementos` cambia el color de fondo del panel de contenido de la ventana.

```java
// Fig. 22.7: MarcoContextual.java
// Demostración de los objetos JPopupMenu.
import java.awt.Color;
import java.awt.event.MouseAdapter;
import java.awt.event.MouseEvent;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import javax.swing.JFrame;
import javax.swing.JRadioButtonMenuItem;
import javax.swing.JPopupMenu;
import javax.swing.ButtonGroup;

public class MarcoContextual extends JFrame
{
```

Figura 22.7 | Objeto `JPopupMenu` para seleccionar colores. (Parte 1 de 3).

```
   private JRadioButtonMenuItem elementos[]; // contiene los elementos para los colores
   private final Color valoresColores[] =
      { Color.BLUE, Color.YELLOW, Color.RED }; // colores a utilizar
   private JPopupMenu menuContextual; // permite al usuario seleccionar el color

   // el constructor sin argumentos establece la GUI
   public MarcoContextual()
   {
      super( "Uso de objetos JPopupMenu" );

      ManejadorElementos manejador = new ManejadorElementos(); // manejador para los elementos de menú
      String colores[] = { "Azul", "Amarillo", "Rojo" }; // arreglo de colores

      ButtonGroup grupoColores = new ButtonGroup(); // administra los elementos de colores
      menuContextual = new JPopupMenu(); // crea el menú contextual
      elementos = new JRadioButtonMenuItem[ 3 ]; // elementos para seleccionar el color

      // construye elemento de menú, lo agrega al menú contextual, habilita el manejo de eventos
      for ( int cuenta = 0; cuenta < elementos.length; cuenta++ )
      {
         elementos[ cuenta ] = new JRadioButtonMenuItem( colores[ cuenta ] );
         menuContextual.add( elementos[ cuenta ] ); // agrega elemento al menú contextual
         grupoColores.add( elementos[ cuenta ] ); // agrega elemento al grupo de botones
         elementos[ cuenta ].addActionListener( manejador ); // agrega el manejador
      } // fin de for

      setBackground( Color.WHITE ); // establece el color de fondo en blanco

      // declara un objeto MouseListener para que la ventana muestre el menú contextual
      addMouseListener(

         new MouseAdapter() // clase interna anónima
         {
            // maneja el evento de oprimir el botón del ratón
            public void mousePressed( MouseEvent evento )
            {
               checkForTriggerEvent( evento ); // comprueba el desencadenador
            } // fin del método mousePressed

            // maneja el evento de liberación del botón del ratón
            public void mouseReleased( MouseEvent evento )
            {
               checkForTriggerEvent( evento ); // comprueba el desencadenador
            } // fin del método mouseReleased

            // determina si el evento debe desencadenar el menú contextual
            private void checkForTriggerEvent( MouseEvent evento )
            {
               if ( evento.isPopupTrigger() )
                  menuContextual.show(
                     evento.getComponent(), evento.getX(), evento.getY() );
            } // fin del método checkForTriggerEvent
         } // fin de la clase interna anónima
      ); // fin de la llamada a addMouseListener
```

Figura 22.7 | Objeto `JPopupMenu` para seleccionar colores. (Parte 2 de 3).

```
69     } // fin del constructor de MarcoContextual
70
71     // clase interna privada para manejar los eventos de los elementos de menú
72     private class ManejadorElementos implements ActionListener
73     {
74        // procesa las selecciones de los elementos de menú
75        public void actionPerformed( ActionEvent evento )
76        {
77           // determina cuál elemento de menú se seleccionó
78           for ( int i = 0; i < elementos.length; i++ )
79           {
80              if ( evento.getSource() == elementos[ i ] )
81              {
82                 getContentPane().setBackground( valoresColores[ i ] );
83                 return;
84              } // fin de if
85           } // fin de for
86        } // fin del método actionPerformed
87     } // fin de la clase interna privada ManejadorElementos
88  } // fin de la clase MarcoContextual
```

Figura 22.7 | Objeto `JPopupMenu` para seleccionar colores. (Parte 3 de 3).

En la línea 25 del constructor de `MarcoContextual` (líneas 21 a 69) se crea una instancia de la clase `ManejadorElementos` (declarada en las líneas 72 a 87), la cual procesará los eventos de los elementos de menú en el menú contextual. En la línea 29 se crea el objeto `JPopupMenu`. La instrucción `for` (líneas 33 a 39) crea un objeto `JRadioButtonMenuItem` (línea 35), lo agrega al objeto `menuContextual` (línea 36), agrega este objeto al objeto `ButtonGroup` llamado `grupoColores` (línea 37) para mantener sólo un objeto `JRadioButtonMenuItem` seleccionado a la vez, y registra su objeto `ActionListener` (línea 38). En la línea 41 se establece el fondo inicial en blanco, invocando al método `setBackground`.

En las líneas 44 a 68 se registra un objeto `MouseListener` para manejar los eventos de ratón de la ventana de aplicación. Los métodos `mousePressed` (líneas 49 a 52) y `mouseReleased` (líneas 55 a 58) comprueban el evento de desencadenamiento del menú contextual. Cada método llama al método utilitario privado `checkForTriggerEvent` (líneas 61 a 66) para determinar si ocurrió el evento de desencadenamiento del menú contextual. De ser así, el método **`isPopupTrigger`** de `MouseEvent` devuelve `true`, y el método **`show`** de `JPopupMenu` muestra el objeto `JPopupMenu`. El primer argumento del método `show` especifica el **componente de origen**, cuya posición ayuda a determinar en dónde aparecerá objeto `JPopupMenu` en la pantalla. Los últimos dos argumentos son las coordenadas *x-y* (medidas desde la esquina superior izquierda del componente de origen) en donde debe aparecer el objeto `JPopupMenu`.

```
// Fig. 22.8: PruebaContextual.java
// Prueba de MarcoContextual.
import javax.swing.JFrame;

public class PruebaContextual
{
   public static void main( String args[] )
   {
      MarcoContextual marcoContextual = new MarcoContextual(); // crea MarcoContextual
      marcoContextual.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
      marcoContextual.setSize( 300, 200 ); // establece el tamaño del marco
      marcoContextual.setVisible( true ); // muestra el marco
   } // fin de main
} // fin de la clase PruebaContextual
```

Figura 22.8 | Clase de prueba para `MarcoContextual`. (Parte 1 de 2).



Figura 22.8 | Clase de prueba para `MarcoContextual`. (Parte 2 de 2).



Observación de apariencia visual 22.10

Para mostrar un objeto `JPopupMenu` para el evento de desencadenamiento de menú contextual de varios componentes de la GUI, se deben registrar manejadores de eventos de ratón para cada uno de esos componentes de la GUI.

Cuando el usuario selecciona un elemento del menú contextual, el método `actionPerformed` de la clase `ManejadorElementos` (líneas 75 a 86) determina cuál objeto `JRadioButtonMenuItem` fue seleccionado por el usuario y establece el color de fondo del panel de contenido de la ventana.

22.6 Apariencia visual adaptable

Un programa que utiliza componentes de la GUI del Abstract Window Toolkit de Java (paquete `java.awt`) asume la apariencia visual de la plataforma en la que se ejecute. Una aplicación de Java ejecutándose en una Macintosh tiene la misma apariencia que las demás aplicaciones que se ejecutan en la Macintosh. Una aplicación de Java ejecutándose en Microsoft Windows tiene la misma apariencia que las demás aplicaciones que se ejecutan en Microsoft Windows. Una aplicación de Java ejecutándose en una plataforma UNIX tiene la misma apariencia que las demás aplicaciones que se ejecutan en esa plataforma UNIX. Esto puede ser conveniente, ya que permite a los usuarios del programa utilizar, en cada plataforma, los componentes de la GUI con los que ya están familiarizados. Sin embargo, esto también introduce algunas cuestiones interesantes, relacionadas con la portabilidad.



Tip de portabilidad 22.1

Los componentes de la GUI en cada plataforma tienen distinta apariencia, lo cual puede requerir de distintas cantidades de espacio para mostrarse en pantalla. Esto podría cambiar la distribución y alineación de los componentes de la GUI.



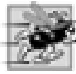
Tip de portabilidad 22.2

Los componentes de la GUI en cada plataforma tienen distinta funcionalidad predeterminada (por ejemplo, algunas plataformas permiten que un botón con el foco se "oprima" mediante la barra de espaciamiento, mientras que otras no).

Los componentes de la GUI ligeros de Swing eliminan muchas de estas cuestiones, al proporcionar una funcionalidad uniforme entre plataformas, y al definir una apariencia visual uniforme entre plataformas (a la que se le conoce como la **apariencia visual metálica**). Swing también proporciona la flexibilidad de personalizar la apariencia visual, para que un programa tenga la apariencia visual al estilo Microsoft Windows (en sistemas Windows), al estilo Motif (UNIX) (en todas las plataformas) o al estilo Mac (sistemas Mac).

La aplicación de las figuras 22.9 y 22.10 demuestra cómo cambiar la apariencia visual de una GUI de Swing. La aplicación crea varios componentes de la GUI, por lo que usted podrá ver el cambio en la apariencia visual de varios componentes de la GUI al mismo tiempo. La primera ventana de salida muestra la apariencia visual metálica estándar, la segunda ventana de salida muestra la apariencia visual Motif y la tercera ventana muestra la apariencia visual Windows.

Todos los componentes de la GUI y el manejo de eventos de este ejemplo se han descrito anteriormente, por lo que ahora nos concentraremos en el mecanismo para cambiar la apariencia visual. La clase **`UIManager`** (paquete `javax.swing`) contiene la clase anidada **`LookAndFeelInfo`** (una clase `public static`) que mantiene la información acerca de una apariencia visual. En la línea 22 se declara un arreglo de tipo `UIManager.LookAndFeelInfo` (observe la sintaxis utilizada para identificar a la clase interna `LookAndFeelInfo`). En la línea 68 se usa el método `static` **`getInstalledLookAndFeels`** de `UIManager` para obtener el arreglo de objetos `UIManager.LookAndFeelInfo` que describe cada una de las apariencias visuales disponibles en su sistema.



Tip de rendimiento 22.2

Cada apariencia visual se representa mediante una clase de Java. El método `getInstalledLookAndFeels` de `UIManager` no carga a cada una de esas clases. En vez de ello, proporciona los nombres de las clases de apariencia visual disponibles, de manera que pueda seleccionarse una de ellas (se supone que una vez, al inicio del programa). Esto reduce la sobrecarga que se genera al cargar clases adicionales que el programa no va a utilizar.

```
1  // Fig. 22.9: MarcoAparienciaVisual.java
2  // Cambio de la apariencia visual.
3  import java.awt.GridLayout;
4  import java.awt.BorderLayout;
5  import java.awt.event.ItemListener;
6  import java.awt.event.ItemEvent;
7  import javax.swing.JFrame;
8  import javax.swing.UIManager;
9  import javax.swing.JRadioButton;
10 import javax.swing.ButtonGroup;
11 import javax.swing.JButton;
12 import javax.swing.JLabel;
13 import javax.swing.JComboBox;
14 import javax.swing.JPanel;
15 import javax.swing.SwingConstants;
16 import javax.swing.SwingUtilities;
17 
18 public class MarcoAparienciaVisual extends JFrame
19 {
20    // nombres de las apariencias visuales
21    private final String cadenas[] = { "Metal", "Motif", "Windows" };
22    private UIManager.LookAndFeelInfo apariencias[]; // apariencias visuales
23    private JRadioButton opcion[]; // botones de opción para seleccionar la apariencia
      visual
24    private ButtonGroup grupo; // grupo para los botones de opción
25    private JButton boton; // muestra la apariencia del botón
26    private JLabel etiqueta; // muestra la apariencia de la etiqueta
27    private JComboBox cuadroComb; // muestra la apariencia del cuadro combinado
28 
29    // establece la GUI
30    public MarcoAparienciaVisual()
31    {
32       super( "Demo de apariencia visual" );
33 
34       JPanel panelNorte = new JPanel(); // crea panel norte
35       panelNorte.setLayout( new GridLayout( 3, 1, 0, 5 ) );
36 
37       etiqueta = new JLabel( "Esta es una apariencia visual metalica",
38          SwingConstants.CENTER ); // crea etiqueta
39       panelNorte.add( etiqueta ); // agrega etiqueta al panel
40 
```

Figura 22.9 | Apariencia visual de una GUI basada en Swing. (Parte 1 de 3).

```
      boton = new JButton( "JButton" ); // crea botón
      panelNorte.add( boton ); // agrega botón al panel

      cuadroComb = new JComboBox( cadenas ); // crea cuadro combinado
      panelNorte.add( cuadroComb ); // agrega cuadro combinado al panel

      // crea arreglo para los botones de opción
      opcion = new JRadioButton[ cadenas.length ];

      JPanel panelSur = new JPanel(); // crea panel sur
      panelSur.setLayout( new GridLayout( 1, opcion.length ) );

      grupo = new ButtonGroup(); // grupo de botones para las apariencias visuales
      ManejadorElementos manejador = new ManejadorElementos(); // manejador de apariencia visual

      for ( int cuenta = 0; cuenta < opcion.length; cuenta++ )
      {
         opcion[ cuenta ] = new JRadioButton( cadenas[ cuenta ] );
         opcion[ cuenta ].addItemListener( manejador ); // agrega el manejador
         grupo.add( opcion[ cuenta ] ); // agrega botón de opción al grupo
         panelSur.add( opcion[ cuenta ] ); // agrega botón de opción al panel
      } // fin de for

      add( panelNorte, BorderLayout.NORTH ); // agrega panel norte
      add( panelSur, BorderLayout.SOUTH ); // agrega panel sur

      // obtiene la información de la apariencia visual instalada
      apariencias = UIManager.getInstalledLookAndFeels();
      opcion[ 0 ].setSelected( true ); // establece la selección predeterminada
   } // fin del constructor de MarcoAparienciaVisual

   // usa UIManager para cambiar la apariencia visual de la GUI
   private void cambiarAparienciaVisual( int valor )
   {
      try // cambia la apariencia visual
      {
         // establece la apariencia visual para esta aplicación
         UIManager.setLookAndFeel( apariencias[ valor ].getClassName() );

         // actualiza los componentes en esta aplicación
         SwingUtilities.updateComponentTreeUI( this );
      } // fin de try
      catch ( Exception excepcion )
      {
         excepcion.printStackTrace();
      } // fin de catch
   } // fin del método cambiarAparienciaVisual

   // clase interna privada para manejar los eventos de los botones de opción
   private class ManejadorElementos implements ItemListener
   {
      // procesa la selección de apariencia visual del usuario
      public void itemStateChanged( ItemEvent evento )
      {
         for ( int cuenta = 0; cuenta < opcion.length; cuenta++ )
         {
            if ( opcion[ cuenta ].isSelected() )
            {
```

Figura 22.9 | Apariencia visual de una GUI basada en Swing. (Parte 2 de 3).

```
99                 etiqueta.setText( String.format( "Esta es una apariencia visual %s",
100                   cadenas[ cuenta ] ) );
101                cuadroComb.setSelectedIndex( cuenta ); // establece el índice del cuadro
                   combinado
102                cambiarAparienciaVisual( cuenta ); // cambia la apariencia visual
103             } // fin de if
104          } // fin de for
105       } // fin del método itemStateChanged
106    } // fin de la clase interna privada ManejadorElementos
107 } // fin de la clase MarcoAparienciaVisual
```

Figura 22.9 | Apariencia visual de una GUI basada en Swing. (Parte 3 de 3).

```
// Fig. 22.10: PruebaContextual.java
// Prueba de MarcoContextual.
import javax.swing.JFrame;

public class PruebaContextual
{
   public static void main( String args[] )
   {
      MarcoContextual marcoContextual = new MarcoContextual(); // crea MarcoContextual
      marcoContextual.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
      marcoContextual.setSize( 300, 200 ); // establece el tamaño del marco
      marcoContextual.setVisible( true ); // muestra el marco
   } // fin de main
} // fin de la clase PruebaContextual
```



Figura 22.10 | Clase de prueba para `MarcoContextual`.

Nuestro método utilitario `cambiarAparienciaVisual` (líneas 73 a 87) es llamado por el manejador de eventos para los objetos `JRadioButton` que se encuentran en la parte inferior de la interfaz de usuario. El manejador de eventos (declarado en la clase interna `private ManejadorElementos` en las líneas 90 a 106) le pasa un valor entero que representa el elemento en el arreglo `apariencias` que deberá utilizarse para cambiar la apariencia visual. En la línea 78 se invoca el método `static` **`setLookAndFeel`** de `UIManager` para cambiar

la apariencia visual. El método **getClassName** de la clase `UIManager.LookAndFeelInfo` determina el nombre de la clase de apariencia visual que corresponde al objeto `UIManager.LookAndFeelInfo`. Si la clase de apariencia visual no se ha cargado ya, se cargará como parte de la llamada a `setLookAndFeel`. En la línea 81 se invoca el método `static` **updateComponentTreeUI** de la clase **SwingUtilities** (paquete `javax.swing`) para cambiar la apariencia visual de todos los componentes de GUI adjuntos a su argumento (la instancia `this` de nuestra clase de aplicación `DemoAparienciaVisual`) a la nueva apariencia visual.

22.7 JDesktopPane y JInternalFrame

Muchas de las aplicaciones de hoy en día utilizan una **interfaz de múltiples documentos (MDI)**: una ventana principal (a la que se le conoce comúnmente como la **ventana padre**) que contiene otras ventanas (a las que se les conoce comúnmente como **ventanas hijas**), para administrar varios **documentos** abiertos que se procesan en paralelo. Por ejemplo, muchos programas de correo electrónico le permiten tener varias ventanas abiertas al mismo tiempo, para que usted pueda componer o leer varios mensajes de correo electrónico de manera simultánea. De manera similar, muchos procesadores de palabras permiten al usuario abrir varios documentos en ventanas separadas, para que el usuario pueda alternar entre los documentos sin tener que cerrar el documento actual para abrir otro. La aplicación de las figuras 22.11 y 22.12 demuestra el uso de las clases **JDesktopPane** y **JInternalFrame** de Swing para implementar interfaces de múltiples documentos.

En las líneas 27 a 33 se crean objetos `JMenuBar`, `JMenu` y `JMenuItem`, se agrega el objeto `JMenuItem` al objeto `JMenu`, se agrega el objeto `JMenu` al objeto `JMenuBar` y se establece el objeto `JMenuBar` para la ventana de aplicación. Cuando el usuario selecciona el objeto `JMenuItem nuevoMarco`, la aplicación crea y muestra un nuevo objeto `JInternalFrame` que contiene una imagen.

```
1  // Fig. 22.11: MarcoEscritorio.java
2  // Demostración de JDesktopPane.
3  import java.awt.BorderLayout;
4  import java.awt.Dimension;
5  import java.awt.Graphics;
6  import java.awt.event.ActionListener;
7  import java.awt.event.ActionEvent;
8  import java.util.Random;
9  import javax.swing.JFrame;
10 import javax.swing.JDesktopPane;
11 import javax.swing.JMenuBar;
12 import javax.swing.JMenu;
13 import javax.swing.JMenuItem;
14 import javax.swing.JInternalFrame;
15 import javax.swing.JPanel;
16 import javax.swing.ImageIcon;
17
18 public class MarcoEscritorio extends JFrame
19 {
20    private JDesktopPane elEscritorio;
21
22    // establece la GUI
23    public MarcoEscritorio()
24    {
25       super( "Uso de JDesktopPane" );
26
27       JMenuBar barra = new JMenuBar(); // crea la barra de menús
28       JMenu menuAgregar = new JMenu( "Agregar" ); // crea el menú Agregar
29       JMenuItem nuevoMarco = new JMenuItem( "Marco interno" );
30
31       menuAgregar.add( nuevoMarco ); // agrega nuevo elemento marco al menú Agregar
```

Figura 22.11 | Interfaz de múltiples documentos. (Parte 1 de 2).

```
      barra.add( menuAgregar ); // agrega el menú Agregar a la barra de menús
      setJMenuBar( barra ); // establece la barra de menús para esta aplicación

      elEscritorio = new JDesktopPane(); // crea el panel de escritorio
      add( elEscritorio ); // agrega el panel de escritorio al marco

      // establece componente de escucha para el elemento de menú nuevoMarco
      nuevoMarco.addActionListener(

         new ActionListener() // clase interna anónima
         {
            // muestra la nueva ventana interna
            public void actionPerformed( ActionEvent evento )
            {
               // crea el marco interno
               JInternalFrame marco = new JInternalFrame(
                  "Marco interno", true, true, true, true );

               MiJPanel panel = new MiJPanel(); // crea nuevo panel
               marco.add( panel, BorderLayout.CENTER ); // agrega el panel
               marco.pack(); // establece marco interno al tamaño del contenido

               elEscritorio.add( marco ); // adjunta marco interno
               marco.setVisible( true ); // muestra marco interno
            } // fin del método actionPerformed
         } // fin de la clase interna anónima
      ); // fin de la llamada a addActionListener
   } // fin del constructor de MarcoEscritorio
} // fin de la clase MarcoEscritorio

// clase para mostrar un objeto ImageIcon en un panel
class MiJPanel extends JPanel
{
   private static Random generador = new Random();
   private ImageIcon imagen; // imagen a mostrar
   private String[] imagenes = { "floresamarillas.png", "floresmoradas.png",
      "floresrojas.png", "floresrojas2.png", "floreslavanda.png" };

   // carga la imagen
   public MiJPanel()
   {
      int numeroAleatorio = generador.nextInt( 5 );
      imagen = new ImageIcon( imagenes[ numeroAleatorio ] ); // establece el icono
   } // fin del constructor de MiJPanel

   // muestra el objeto imageIcon en el panel
   public void paintComponent( Graphics g )
   {
      super.paintComponent( g );
      imagen.paintIcon( this, g, 0, 0 ); // muestra el icono
   } // fin del método paintComponent

   // devuelve las medidas de la imagen
   public Dimension getPreferredSize()
   {
      return new Dimension( imagen.getIconWidth(),
         imagen.getIconHeight() );
   } // fin del método getPreferredSize
} // fin de la clase MiJPanel
```

Figura 22.11 | Interfaz de múltiples documentos. (Parte 2 de 2).

```java
// Fig. 22.12: PruebaEscritorio.java
// Demostración de MarcoEscritorio.
import javax.swing.JFrame;

public class PruebaEscritorio
{
   public static void main( String args[] )
   {
      MarcoEscritorio marcoEscritorio = new MarcoEscritorio();
      marcoEscritorio.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
      marcoEscritorio.setSize( 600, 480 ); // establece el tamaño del marco
      marcoEscritorio.setVisible( true ); // muestra el marco
   } // fin de main
} // fin de la clase PruebaEscritorio
```



Figura 22.12 | Clase de prueba para `MarcoEscritorio`.

En la línea 35 se asigna la variable `JDesktopPane` (paquete `javax.swing`) llamada `elEscritorio` a un nuevo objeto `JDesktopPane` que se utilizará para administrar las ventanas hijas `JInternalFrame`. En la línea 36 se agrega el objeto `JDesktopPane` al objeto `JFrame`. De manera predeterminada, el objeto `JDesktopPane` se agrega al centro del esquema `BorderLayout` del panel de contenido, por lo que el objeto `JDesktopPane` se expande para rellenar toda la ventana de aplicación.

En las líneas 39 a 58 se registra un objeto `ActionListener` para manejar el evento cuando el usuario selecciona el elemento de menú `nuevoMarco`. Cuando ocurre el evento, el método `actionPerformed` (líneas 44 a 56) crea un objeto `JInternalFrame` en las líneas 47 y 48. El constructor de `JInternalFrame` que se utiliza aquí requiere cinco argumentos: una cadena para la barra de título de la ventana interna, un valor `boolean` que indique si el usuario puede reajustar el tamaño del marco interno, un valor `boolean` que indique si el usuario puede cerrar el marco interno, un valor `boolean` que indique si el usuario puede maximizar el marco interno y un valor `boolean` que indique si el usuario puede minimizar el marco interno. Para cada uno de los argumentos `boolean`, un valor de `true` indica que la operación debe permitirse (como se da el caso aquí).

Al igual que con los objetos `JFrame` y `JApplet`, un objeto `JInternalFrame` tiene un panel de contenido, al cual pueden adjuntarse componentes de la GUI. En la línea 50 se crea una instancia de nuestra clase `MiJPanel` (declarada en las líneas 63 a 90), la cual se agrega al objeto `JInternalFrame` en la línea 51.

En la línea 52 se utiliza el método **`pack`** de `JInternalFrame` para establecer el tamaño de la ventana hija. El método `pack` utiliza los tamaños preferidos de los componentes para determinar el tamaño de la ventana. La clase `MiJPanel` declara el método **`getPreferredSize`** (líneas 85 a 89) para especificar el tamaño preferido del panel, para que lo use el método `pack`. En la línea 54 se agrega el objeto `JInternalFrame` al objeto `JDesktopPane`, y en la línea 55 se muestra el objeto `JInternalFrame`.

Las clases `JInternalFrame` y `JDesktopPane` proporcionan muchos métodos para administrar ventanas hijas. Vea la documentación de la API en línea acerca de `JInternalFrame` y `JDesktopPane`, para obtener listas completas de estos métodos:

```
java.sun.com/javase/6/docs/api/javax/swing/JInternalFrame.html
java.sun.com/javase/6/docs/api/javax/swing/JDesktopPane.html
```

22.8 JTabbedPane

Un objeto **`JTabbedPane`** ordena los componentes de la GUI en capas, en donde sólo una capa está visible en un momento dado. Los usuarios acceden a cada una de las capas mediante una ficha; algo muy parecido a las carpetas en un archivero. Cuando el usuario hace clic en una ficha, se muestra la capa apropiada. Las fichas aparecen en la parte superior de manera predeterminada, pero también pueden colocarse a la izquierda, derecha o en la parte inferior del objeto `JTabbedPane`. Puede colocarse cualquier componente en una ficha. Si el componente es un contenedor como un panel, puede utilizar cualquier administrador de esquemas para distribuir varios componentes en esa ficha. La clase `JTabbedPane` es una subclase de `JComponent`. La aplicación de las figuras 22.13 y 22.14 crea un panel con tres fichas. Cada ficha muestra uno de los objetos `JPanel`: `panel1`, `panel2` o `panel3`.

```
// Fig. 22.13: MarcoJTabbedPane.java
// Demostración de JTabbedPane.
import java.awt.BorderLayout;
import java.awt.Color;
import javax.swing.JFrame;
import javax.swing.JTabbedPane;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JButton;
import javax.swing.SwingConstants;

public class MarcoJTabbedPane extends JFrame
{
   // establece la GUI
```

Figura 22.13 | Uso de un objeto `JTabbedPane` para organizar los componentes de una GUI. (Parte 1 de 2).

```
15      public MarcoJTabbedPane()
16      {
17         super( "Demo de JTabbedPane " );
18
19         JTabbedPane panelFichas = new JTabbedPane(); // crea objeto JTabbedPane
20
21         // establece pane11 y lo agrega al objeto JTabbedPane
22         JLabel etiqueta1 = new JLabel( "panel uno", SwingConstants.CENTER );
23         JPanel panel1 = new JPanel(); // crea el primer panel
24         panel1.add( etiqueta1 ); // agrega etiqueta al panel
25         panelFichas.addTab( "Ficha uno", null, panel1, "Primer panel" );
26
27         // establece panel2 y lo agrega al objeto JTabbedPane
28         JLabel etiqueta2 = new JLabel( "panel dos", SwingConstants.CENTER );
29         JPanel panel2 = new JPanel(); // crea el segundo panel
30         panel2.setBackground( Color.YELLOW ); // establece el color de fondo en amarillo
31         panel2.add( etiqueta2 ); // agrega etiqueta al panel
32         panelFichas.addTab( "Ficha dos", null, panel2, "Segundo panel" );
33
34         // establece panel3 y lo agrega al objeto JTabbedPane
35         JLabel etiqueta3 = new JLabel( "panel tres" );
36         JPanel panel3 = new JPanel(); // crea el tercer panel
37         panel3.setLayout( new BorderLayout() ); // usa esquema Borderlayout
38         panel3.add( new JButton( "Norte" ), BorderLayout.NORTH );
39         panel3.add( new JButton( "Oeste" ), BorderLayout.WEST );
40         panel3.add( new JButton( "Este" ), BorderLayout.EAST );
41         panel3.add( new JButton( "Sur" ), BorderLayout.SOUTH );
42         panel3.add( etiqueta3, BorderLayout.CENTER );
43         panelFichas.addTab( "Ficha tres", null, panel3, "Tercer panel" );
44
45         add( panelFichas ); // agrega objeto JTabbedPane al marco
46      } // fin del constructor de MarcoJTabbedPane
47   } // fin de la clase MarcoJTabbedPane
```

Figura 22.13 | Uso de un objeto `JTabbedPane` para organizar los componentes de una GUI. (Parte 2 de 2).

El constructor (líneas 15 a 46) crea la GUI. En la línea 19 se crea un objeto `JTabbedPane` vacío con la configuración predeterminada (es decir, fichas en la parte superior). Si las fichas no se ajustan en una línea, pasarán a la siguiente para formar líneas adicionales de fichas. A continuación, el constructor crea los objetos `JPanel` `panel1`, `panel2` y `panel3`, junto con sus componentes de la GUI. A medida que configuramos cada panel, lo agregamos al objeto `panelConFichas` utilizando el método **addTab** de `JTabbedPane` con cuatro argumentos. El primer argumento es una cadena que especifica el título de la ficha. El segundo es una referencia `Icon` que especifica un icono a mostrar en la ficha. Si el objeto `Icon` es una referencia `null`, no se muestra una imagen. El tercer argumento es una referencia `Component` que representa el componente de la GUI a mostrar cuando el usuario hace clic en la ficha. El último argumento es una cadena que especifica el cuadro de información sobre herramientas de la ficha. Por ejemplo, en la línea 25 se agrega el objeto `JPanel` `panel1` al objeto `panelFichas` con el título `"Ficha uno"` y la información sobre herramientas `"Primer panel"`. Los objetos `JPanel` `panel2` y `panel3` se agregan a `panelFichas` en las líneas 32 y 43. Para ver cada una de las fichas, haga clic en la ficha deseada con el ratón o utilice las teclas de dirección para avanzar por cada una de las fichas.

```
// Fig. 22.14: DemoJTabbedPane.java
// Demostración de JTabbedPane.
import javax.swing.JFrame;

```

Figura 22.14 | Clase de prueba para `MarcoJTabbedPane`. (Parte 1 de 2).



```
public class DemoJTabbedPane
{
   public static void main( String args[] )
   {
      MarcoJTabbedPane marcoPanelFichas = new MarcoJTabbedPane();
      marcoPanelFichas.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
      marcoPanelFichas.setSize( 250, 200 ); // establece el tamaño del marco
      marcoPanelFichas.setVisible( true ); // muestra el marco
   } // fin de main
} // fin de la clase DemoJTabbedPane
```


Figura 22.14 | Clase de prueba para `MarcoJTabbedPane`. (Parte 2 de 2).

22.9 Administradores de esquemas: BoxLayout y GridBagLayout

En el capítulo 11 presentamos tres administradores de esquemas: `FlowLayout`, `BorderLayout` y `GridLayout`. En esta sección presentamos dos administradores de esquemas adicionales (los cuales se sintetizan en la figura 22.15). En los siguientes ejemplos, hablaremos sobre estos administradores de esquemas.

Administrador de esquemas	Descripción
`BoxLayout`	Un administrador de esquemas que permite ordenar los componentes de la GUI de izquierda a derecha, o de arriba hacia abajo, en un contenedor. La clase `Box` declara un contenedor con `BoxLayout` como su administrador de esquemas predeterminado, y proporciona métodos estáticos para crear un objeto `Box` con un esquema `BoxLayout` horizontal o vertical.
`GridBagLayout`	Un administrador de esquemas similar a `GridLayout`. A diferencia de `GridLayout`, el tamaño de cada componente puede variar y pueden agregarse componentes en cualquier orden.

Figura 22.15 | Administradores de esquemas adicionales.

Administrador de esquemas **BoxLayout**

El administrador de esquemas `BoxLayout` (en el paquete `javax.swing`) ordena los componentes de la GUI en forma horizontal a lo largo del eje *x*, o en forma vertical a lo largo del eje *y* de un contenedor. La aplicación de las figuras 22.16 a 22.17 demuestra el uso del esquema `BoxLayout` y la clase contenedora `Box` que utiliza a `BoxLayout` como su administrador de esquemas predeterminado.

En las líneas 19 a 22 se crean contenedores `Box`. Las referencias `horizontal1` y `horizontal2` se inicializan con el método estático `createHorizontalBox` de `Box`, el cual devuelve un contenedor `Box` con un esquema `BoxLayout` horizontal en el que los componentes de la GUI se ordenan de izquierda a derecha. Las variables `vertical1` y `vertical2` se inicializan con el método estático **`createVerticalBox`** de `Box`, el cual devuelve refe-

```
// Fig. 22.16: MarcoBoxLayout.java
// Demostración de BoxLayout.
import java.awt.Dimension;
import javax.swing.JFrame;
import javax.swing.Box;
import javax.swing.JButton;
import javax.swing.BoxLayout;
import javax.swing.JPanel;
import javax.swing.JTabbedPane;

public class MarcoBoxLayout extends JFrame
{
   // establece la GUI
   public MarcoBoxLayout()
   {
      super( "Demostración de BoxLayout" );

      // crea contenedores Box con BoxLayout
      Box horizontal1 = Box.createHorizontalBox();
      Box vertical1 = Box.createVerticalBox();
      Box horizontal2 = Box.createHorizontalBox();
      Box vertical2 = Box.createVerticalBox();

      final int TAMANIO = 3; // número de botones en cada objeto Box

      // agrega botones al objeto Box horizontal1
      for ( int cuenta = 0; cuenta < TAMANIO; cuenta++ )
         horizontal1.add( new JButton( "Boton " + cuenta ) );

      // crea montante y agrega botones al objeto Box vertical1
      for ( int cuenta = 0; cuenta < TAMANIO; cuenta++ )
      {
         vertical1.add( Box.createVerticalStrut( 25 ) );
         vertical1.add( new JButton( "Boton " + cuenta ) );
      } // fin de for

      // crea pegamento horizontal y agrega botones al objeto Box horizontal2
      for ( int cuenta = 0; cuenta < TAMANIO; cuenta++ )
      {
         horizontal2.add( Box.createHorizontalGlue() );
         horizontal2.add( new JButton( "Boton " + cuenta ) );
      } // fin de for

      // crea un área rígida y agrega botones al objeto Box vertical2
      for ( int cuenta = 0; cuenta < TAMANIO; cuenta++ )
      {
         vertical2.add( Box.createRigidArea( new Dimension( 12, 8 ) ) );
         vertical2.add( new JButton( "Boton " + cuenta ) );
      } // fin de for

      // crea pegamento vertical y agrega botones al panel
      JPanel panel = new JPanel();
      panel.setLayout( new BoxLayout( panel, BoxLayout.Y_AXIS ) );

      for ( int cuenta = 0; cuenta < TAMANIO; cuenta++ )
      {
         panel.add( Box.createGlue() );
         panel.add( new JButton( "Boton " + cuenta ) );
      } // fin de for
```

Figura 22.16 | Administrador de esquemas BoxLayout. (Parte 1 de 2).

```
60
61         // crea un objeto JTabbedPane
62         JTabbedPane fichas = new JTabbedPane(
63            JTabbedPane.TOP, JTabbedPane.SCROLL_TAB_LAYOUT );
64
65         // coloca cada contenedor en el panel con fichas
66         fichas.addTab( "Cuadro horizontal", horizontal1 );
67         fichas.addTab( "Cuadro vertical con montantes", vertical1 );
68         fichas.addTab( "Cuadro horizontal con pegamento", horizontal2 );
69         fichas.addTab( "Cuadro vertical con areas rigidas", vertical2 );
70         fichas.addTab( "Cuadro vertical con pegamento", panel );
71
72         add( fichas ); // coloca panel con fichas en el marco
73      } // fin del constructor de MarcoBoxLayout
74   } // fin de la clase MarcoBoxLayout
```

Figura 22.16 | Administrador de esquemas BoxLayout. (Parte 2 de 2).

rencias a contenedores Box con un esquema BoxLayout vertical, en el que los componentes de la GUI se ordenan de arriba hacia abajo.

La instrucción for de las líneas 27 y 28 agrega tres objetos JButton a horizontal1. La instrucción for de las líneas 31 a 35 agrega tres objetos JButton a vertical1. Antes de agregar cada botón, en la línea 33 se agrega un **montante vertical** al contenedor, mediante el método estático **createVerticalStrut** de Box. Un montante vertical es un componente de la GUI invisible, el cual tiene una altura fija en píxeles y se utiliza para garantizar una cantidad fija de espacio entre los componentes de la GUI. El argumento int para el método createVerticalStrut determina la altura del montante, en píxeles. Cuando se cambia el tamaño del contenedor, la distancia entre los componentes de la GUI que están separados por montantes no cambia. La clase Box también declara el método **createHorizontalStrut** para esquemas BoxLayout horizontales.

La instrucción for de las líneas 38 a 42 agrega tres objetos JButton a horizontal2. Antes de agregar cada botón, en la línea 40 se agrega **pegamento horizontal** al contenedor, mediante el método estático **createHorizontalGlue** de Box. El pegamento horizontal es un componente de la GUI invisible que puede usarse entre los componentes de la GUI de tamaño fijo para ocupar espacio adicional. Generalmente, el espacio adicional aparece a la derecha del último componente de la GUI horizontal, o debajo del último componente de la GUI vertical, en un esquema BoxLayout. El pegamento permite que se agregue espacio adicional entre los componentes de la GUI. Cuando se cambia el tamaño del contenedor, los componentes separados por pegamento conservan el mismo tamaño, pero el pegamento se estira o se contrae para ocupar el espacio entre los demás componentes. La clase Box también declara el método **createVerticalGlue** para esquemas BoxLayout verticales.

La instrucción for de las líneas 45 a 49 agrega tres objetos JButton a vertical2. Antes de agregar cada botón, en la línea 47 se agrega un **área rígida** al contenedor, mediante el método static **createRigidArea** de Box. Un área rígida es un componente de la GUI invisible, el cual siempre tiene una anchura y altura fijas, en píxeles. El argumento para el método createRigidArea es un objeto Dimension que especifica la anchura y la altura del área rígida.

En las líneas 52 y 53 se crea un objeto JPanel y se establece su esquema en BoxLayout de la manera convencional, utilizando el método setLayout de Container. El constructor de BoxLayout recibe una referencia al contenedor para el que está controlando el esquema, y una constante que indica si el esquema es horizontal (**BoxLayout.X_AXIS**) o vertical (**BoxLayout.Y_AXIS**).

La instrucción for de las líneas 55 a 59 agrega tres objetos JButton al objeto panel. Antes de agregar cada botón, en la línea 57 se agrega un componente pegamento al contenedor, mediante el método static **createGlue** de Box. Este componente se expande o se contrae con base en el tamaño del objeto Box.

En las líneas 62 y 63 se crea un objeto JTabbedPane para mostrar los cinco contenedores en este programa. El argumento **JTabbedPane.TOP** que se envía al constructor indica que las fichas deben aparecer en la parte superior del objeto JTabbedPane. El argumento **JTabbedPane.SCROLL_TAB_LAYOUT** especifica que las fichas deben desplazarse si hay demasiadas como para que puedan ajustarse en una sola línea.

```java
// Fig. 22.17: DemoBoxLayout.java
// Demostración de BoxLayout.
import javax.swing.JFrame;

public class DemoBoxLayout
{
   public static void main( String args[] )
   {
      MarcoBoxLayout marcoBoxLayout = new MarcoBoxLayout();
      marcoBoxLayout.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
      marcoBoxLayout.setSize( 400, 220 ); // establece el tamaño del marco
      marcoBoxLayout.setVisible( true ); // muestra el marco
   } // fin de main
} // fin de la clase DemoBoxLayout
```



Figura 22.17 | Clase de prueba para `MarcoBoxLayout.java` .

Los contenedores `Box` y el objeto `JPanel` se adjuntan al objeto `JTabbedPane` en las líneas 66 a 70. Ahora pruebe a ejecutar la aplicación. Cuando aparezca la ventana, cambie su tamaño para ver cómo afectan los componentes pegamento, montante y área rígida en la distribución de cada ficha.

Administrador de esquemas **GridBagLayout**

Uno de los administradores de esquemas predefinidos más complejos y poderosos es **GridBagLayout** (en el paquete java.awt). Este esquema es similar a GridLayout, ya que también ordena los componentes en una cuadrícula. Sin embargo, el esquema GridBagLayout es más flexible. Los componentes pueden variar en tamaño (es decir, pueden ocupar varias filas y columnas) y pueden agregarse en cualquier orden.

El primer paso para utilizar GridBagLayout es determinar la apariencia de la GUI. Para este paso sólo se necesita un pedazo de papel. Dibuje la GUI y después dibuje una cuadrícula sobre la GUI, dividiendo los componentes en filas y columnas. Los números iniciales de fila y columna deben ser 0, de manera que el esquema GridBagLayout pueda usar los números de fila y columna para colocar apropiadamente los componentes en la cuadrícula. En la figura 22.18 se demuestra cómo dibujar las líneas para las filas y columnas sobre una GUI.

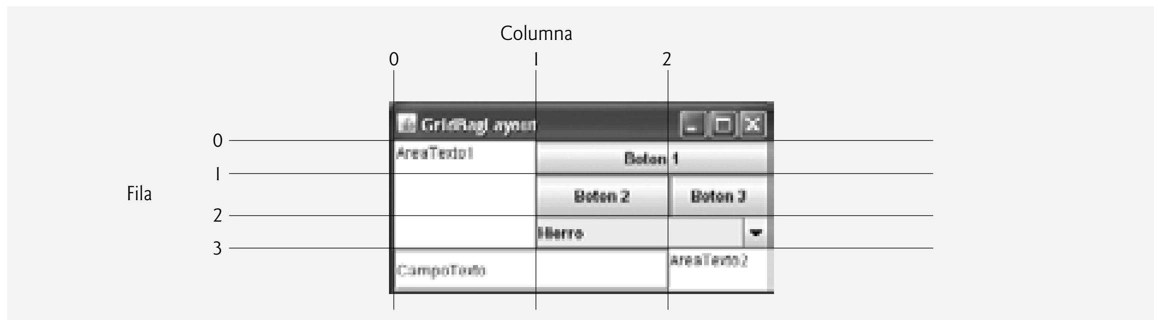


Figura 22.18 | Diseño de una GUI que utilizará a GridBagLayout.

Un objeto **GridBagConstraints** describe cómo colocar un componente en un esquema GridBagLayout. Varios campos de GridBagConstraints se sintetizan en la figura 22.19.

El campo **anchor** de GridBagConstraints especifica la posición relativa del componente en un área que no rellena. A la variable anchor se le asigna una de las siguientes constantes de GridBagConstraints: **NORTH**, **NORTHEAST**, **EAST**, **SOUTHEAST**, **SOUTH**, **SOUTHWEST**, **WEST**, **NORTHWEST** o **CENTER**. El valor predeterminado es CENTER.

Campo de GridBagConstraints	Descripción
anchor	Especifica la posición relativa (NORTH, NORTHEAST, EAST, SOUTHEAST, SOUTH, SOUTHWEST, WEST, NORTHWEST, CENTER) del componente en un área que no rellena.
fill	Ajusta el tamaño del componente en la dirección especificada (NONE, HORIZONTAL, VERTICAL, BOTH) cuando el área en pantalla es más grande que el componente.
gridx	La columna en la que se colocará el componente.
gridy	La fila en la que se colocará el componente.
gridwidth	El número de columnas que ocupa el componente.
gridheight	El número de filas que ocupa el componente.
weightx	La porción de espacio adicional que se asignará horizontalmente. La ranura de la cuadrícula puede hacerse más ancha cuando haya espacio adicional disponible.
weighty	La porción de espacio adicional que se asignará verticalmente. La ranura de la cuadrícula puede hacerse más alta cuando haya espacio adicional disponible.

Figura 22.19 | Campos de GridBagConstraints.

El campo `fill` de `GridBagConstraints` define la forma en que crece el componente, si el área en la que puede mostrarse es mayor que el componente. A la variable `fill` se le asigna una de las siguientes constantes de `GridBagConstraints`: **`NONE`**, **`VERTICAL`**, **`HORIZONTAL`** o **`BOTH`**. El valor predeterminado es `NONE`, el cual indica que el componente no crecerá en ninguna dirección. `VERTICAL` indica que crecerá en forma vertical. `HORIZONTAL` indica que crecerá en forma horizontal. `BOTH` indica que crecerá en ambas direcciones.

Las variables **`gridx`** y **`gridy`** especifican la posición que tendrá la esquina superior izquierda del componente en la cuadrícula. La variable `gridx` corresponde a la columna, y la variable `gridy` corresponde a la fila. En la figura 22.18, el objeto `JComboBox` (que muestra la cadena "`Hierro`") tiene un valor de `gridx` de 1 y un valor de `gridy` de 2.

La variable **`gridwidth`** especifica el número de columnas que ocupa un componente. El objeto `JComboBox` ocupa dos columnas. La variable **`gridheight`** especifica el número de filas que ocupa un componente. El objeto `JTextArea` del lado izquierdo de la figura 22.18 ocupa tres filas.

La variable **`weightx`** especifica cómo distribuir el espacio horizontal adicional en las ranuras de cuadrícula de un esquema `GridBagLayout`, cuando se ajusta el tamaño del contenedor. Un valor de cero indica que la ranura de la cuadrícula no crecerá horizontalmente por sí sola. No obstante, si el componente abarca una columna que contenga un componente con un valor de `weightx` distinto de cero, el componente con valor de `weightx` de cero crecerá horizontalmente en la misma proporción que el (los) otro(s) componente(s) en la misma columna. Esto se debe a que cada componente debe mantenerse en las mismas fila y columna en las que se colocó originalmente.

La variable **`weighty`** especifica cómo distribuir el espacio vertical adicional en las ranuras de cuadrícula de un esquema `GridBagLayout`, cuando se ajusta el tamaño del contenedor. Un valor de cero indica que la ranura de la cuadrícula no crecerá verticalmente por sí sola. No obstante, si el componente abarca una columna que contenga un componente con un valor de `weighty` distinto de cero, el componente con valor de `weighty` de cero crecerá verticalmente en la misma proporción que el (los) otro(s) componente(s) en la misma columna.

En la figura 22.18, los efectos de `weighty` y `weightx` no podrán verse fácilmente sino hasta que se ajuste el tamaño del contenedor y haya más espacio adicional disponible. Los componentes con valores mayores para `weightx` y `weighty` ocupan más espacio adicional que los componentes con valores menores.

Los componentes deben recibir valores positivos distintos de cero para `weightx` y `weighty`; de lo contrario, se "amontonarán" al centro del contenedor. En la figura 22.20 se muestra la GUI de la figura 22.18, con un valor de cero para `weightx` y `weighty`.

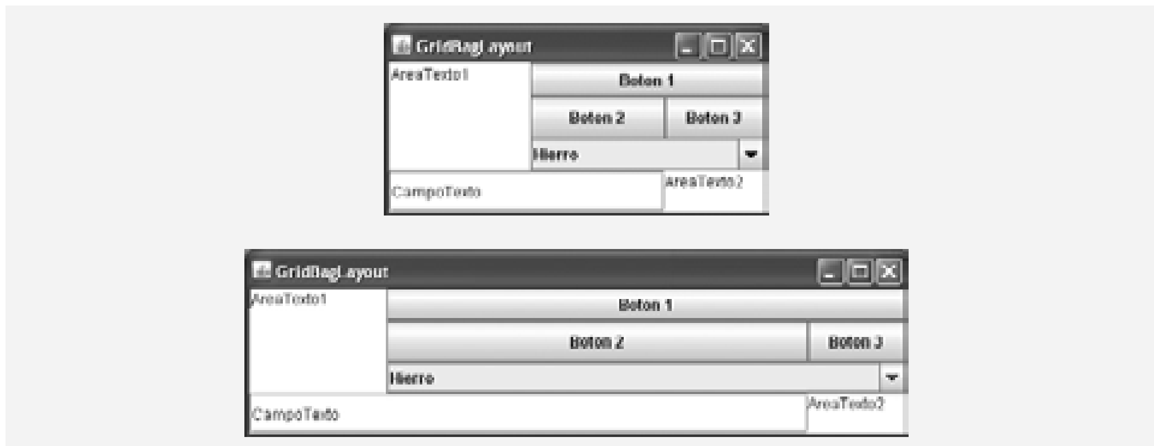


Figura 22.20 | El `GridBagLayout` con los valores establecidos en 0.

La aplicación de las figuras 22.21 y 22.2 utiliza el administrador de esquemas `GridBagLayout` para ordenar los componentes que se encuentran en la GUI de la figura 22.18. Este programa no hace nada más que demostrar cómo utilizar el esquema `GridBagLayout`.

La GUI consiste de tres objetos `JButton`, dos objetos `JTextArea`, un objeto `JComboBox` y un objeto `JTextField`. El administrador de esquemas para el panel de contenido es `GridBagLayout`. En las líneas 21 y 22 se crea

```
// Fig. 22.21: MarcoGridBag.java
// Demostración de GridBagLayout.
import java.awt.GridBagLayout;
import java.awt.GridBagConstraints;
import java.awt.Component;
import javax.swing.JFrame;
import javax.swing.JTextArea;
import javax.swing.JTextField;
import javax.swing.JButton;
import javax.swing.JComboBox;

public class MarcoGridBag extends JFrame
{
   private GridBagLayout esquema; // esquema de este marco
   private GridBagConstraints restricciones; // restricciones de este esquema

   // establece la GUI
   public MarcoGridBag()
   {
      super( "GridBagLayout" );
      esquema = new GridBagLayout();
      setLayout( esquema ); // establece el esquema del marco
      restricciones = new GridBagConstraints(); // instancia las restricciones

      // crea los componentes de la GUI
      JTextArea areaTexto1 = new JTextArea( "AreaTexto1", 5, 10 );
      JTextArea areaTexto2 = new JTextArea( "AreaTexto2", 2, 2 );

      String nombres[] = { "Hierro", "Acero", "Bronce" };
      JComboBox cuadroComb = new JComboBox( nombres );

      JTextField campoTexto = new JTextField( "CampoTexto" );
      JButton boton1 = new JButton( "Boton 1" );
      JButton boton2 = new JButton( "Boton 2" );
      JButton boton3 = new JButton( "Boton 3" );

      // weightx y weighty para areaTexto1 son 0: el valor predeterminado
      // anchor para todos los componentes es CENTER: el valor predeterminado
      restricciones.fill = GridBagConstraints.BOTH;
      agregarComponente( areaTexto1, 0, 0, 1, 3 );

      // weightx y weighty para boton1 son 0: el valor predeterminado
      restricciones.fill = GridBagConstraints.HORIZONTAL;
      agregarComponente( boton1, 0, 1, 2, 1 );

      // weightx y weighty para cuadroComb son 0: el valor predeterminado
      // fill es HORIZONTAL
      agregarComponente( cuadroComb, 2, 1, 2, 1 );

      // boton2
      restricciones.weightx = 1000;  // puede hacerse más ancho
      restricciones.weighty = 1;    // puede hacerse más alto
      restricciones.fill = GridBagConstraints.BOTH;
      agregarComponente( boton2, 1, 1, 1, 1 );

      // fill es BOTH para boton3
      restricciones.weightx = 0;
      restricciones.weighty = 0;
      agregarComponente( boton3, 1, 2, 1, 1 );
```

Figura 22.21 | Administrador de esquemas `GridBagLayout`. (Parte 1 de 2).

```
60
61          // weightx y weighty para campoTexto son 0, fill es BOTH
62          agregarComponente( campoTexto, 3, 0, 2, 1 );
63
64          // weightx y weighty para areaTexto2 son 0, fill es BOTH
65          agregarComponente( areaTexto2, 3, 2, 1, 1 );
66       } // fin del constructor de MarcoGridBag
67
68       // método para establecer restricciones
69       private void agregarComponente( Component componente,
70          int fila, int columna, int anchura, int altura )
71       {
72          restricciones.gridx = columna; // establece gridx
73          restricciones.gridy = fila; // establece gridy
74          restricciones.gridwidth = anchura; // establece gridwidth
75          restricciones.gridheight = altura; // establece gridheight
76          esquema.setConstraints( componente, restricciones ); // establece restricciones
77          add( componente ); // agrega el componente
78       } // fin del método agregarComponente
79    } // fin de la clase MarcoGridBag
```

Figura 22.21 | Administrador de esquemas `GridBagLayout`. (Parte 2 de 2).

el objeto `GridBagLayout` y se establece el administrador de esquemas para el panel de contenido en `esquema`. En la línea 23 se crea el objeto `GridBagConstraints` utilizado para determinar la ubicación y el tamaño de cada uno de los componentes en la cuadrícula. En las líneas 26 a 35 se crea cada uno de los componentes de la GUI que se agregarán al panel de contenido.

En las líneas 39 a 40 se configura el objeto `JTextArea areaTexto1` y se agrega al panel de contenido. Los valores para `weightx` y `weighty` no se especifican en `restricciones`, por lo que cada una de ellas tiene el valor de cero, de manera predeterminada. Por lo tanto, el objeto `JTextArea` no se cambiará de tamaño a sí mismo, incluso si hay espacio disponible. Sin embargo, el objeto `JTextArea` abarca varias filas por lo que el tamaño vertical está sujeto a los valores de `weighty` de los objetos `JButton boton2` y `boton3`. Cuando se cambie el tamaño de uno de los objetos `boton2` o `boton3` en forma vertical, con base en su valor de `weighty`, también se cambiará el tamaño del objeto `JTextArea`.

En la línea 39 se establece la variable `fill` de `restricciones` en `GridBagConstraints.BOTH`, lo cual hará que el objeto `JTextArea` siempre llene toda su área asignada en la cuadrícula. No se especifica un valor para `anchor` en `restricciones`, por lo que se utiliza el valor predeterminado de `CENTER`. Como no utilizamos la variable `anchor` en esta aplicación, todos los componentes utilizarán su valor predeterminado. En la línea 40 se hace una llamada a nuestro método utilitario `agregarComponente` (declarado en las líneas 69 a 78). El objeto `JTextArea`, la fila, la columna, el número de columnas y el número de filas a abarcar se pasan como argumentos.

El objeto `JButton boton1` es el siguiente componente que se agrega (líneas 43 y 44). De manera predeterminada, los valores de `weightx` y `weighty` siguen siendo cero. La variable `fill` se establece en `HORIZONTAL`; el componente siempre ocupará toda su área en dirección horizontal. La dirección vertical no se ocupará toda. El valor de `weighty` es cero, por lo que el botón sólo será más alto si otro componente en la misma fila tiene un valor de `weighty` distinto de cero. El objeto `JButton boton1` se encuentra en la fila `0`, columna `1`. Se ocupan una fila y dos columnas.

El objeto `JComboBox cuadroCombinado` es el siguiente componente que se agrega (línea 48). De manera predeterminada, los valores de `weightx` y `weighty` son cero y la variable `fill` se establece en `HORIZONTAL`. El botón `JComboBox` crecerá solamente en dirección horizontal. Observe que las variables `weightx`, `weighty` y `fill` retienen los valores establecidos en `restricciones` hasta que se modifiquen. El botón `JComboBox` se coloca en la fila 2, columna 1. Se ocupan una fila y dos columnas.

El objeto `JButton boton2` es el siguiente componente que se agrega (líneas 51 a 54). Obtiene un valor de `weightx` de `1000` y un valor de `weighty` de `1`. El área ocupada por el botón es capaz de crecer en las direcciones vertical y horizontal. La variable `fill` se establece en `BOTH`, lo cual especifica que el botón siempre ocupará toda el área. Cuando se ajuste el tamaño de la ventana, `boton2` crecerá. El botón se coloca en la fila 1, columna 1. Se ocupan una fila y una columna.

```
// Fig. 22.22: DemoGridBag.java
// Demostración de MarcoBagLayout.
import javax.swing.JFrame;

public class DemoGridBag
{
    public static void main( String args[] )
    {
      MarcoGridBag marcoGridBag = new MarcoGridBag();
      marcoGridBag.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
      marcoGridBag.setSize( 300, 150 ); // establece el tamaño del marco
      marcoGridBag.setVisible( true ); // muestra el marco
    } // fin de main
} // fin de la clase DemoGridBag
```





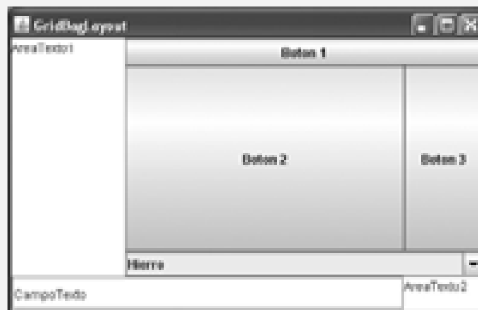


Figura 22.22 | Clase de prueba para `MarcoGridBag`.

El objeto `JButton boton3` se agrega a continuación (líneas 57 a 59). Los valores de `weightx` y `weighty` son cero, y la variable `fill` se establece en `BOTH`. El objeto `JButton boton3` crecerá si se ajusta el tamaño de la ventana; se verá afectado por los valores de `weightx` y `weighty` de `boton2`. Observe que el valor de `weightx` para `boton2` es mucho mayor que el de `boton3`. Cuando ocurra el cambio de tamaño, `boton2` ocupará un mayor porcentaje del nuevo espacio. El botón se coloca en la fila 1, columna 2. Se ocupan una fila y una columna.

Tanto el objeto `JTextField campoTexto` (línea 62) como el objeto `JTextArea areaTexto2` (línea 65) tienen un valor de `weightx` de 0 y un valor de `weighty` de `0`. El valor de `fill` es `BOTH`. El objeto `JTextField` se coloca en la fila 3, columna 0, y el objeto `JTextArea` se coloca en la fila 3, columna 2. El objeto `JTextField` ocupa una fila y dos columnas. El objeto `JTextArea` ocupa una fila y una columna.

Los parámetros del método `agregarComponente` son una referencia `Component` llamada `componente`, y los enteros `fila`, `columna`, `anchura` y `altura`. En las líneas 72 y 73 se establecen las variables de `GridBagConstraints` llamadas `gridx` y `gridy`. A la variable `gridx` se le asigna la columna en la que se colocará el objeto

Component, y a la variable gridy se le asigna la fila en la que se colocará el objeto Component. En las líneas 74 y 75 se establecen las variables de GridBagConstraints llamadas gridwidth y gridheight. La variable gridwidth especifica el número de columnas que abarcará el objeto Component en la cuadrícula, y la variable gridheight especifica el número de filas que abarcará el objeto Component en la cuadrícula. En la línea 76 se establecen los valores del objeto GridBagConstraints para un componente en el esquema GridBagLayout. El método **setConstraints** de la clase GridBagLayout recibe un argumento Component y un argumento GridBagConstraints. En la línea 77 se agrega el componente al objeto JFrame.

Cuando ejecute esta aplicación, pruebe a cambiar el tamaño de la ventana para ver cómo las restricciones para cada componente de la GUI afectan su posición y tamaño en la ventana.

Las constantes **RELATIVE** *y* **REMAINDER** *de* **GridBagConstraints**

Hay una variación de GridBagLayout que no utiliza a las variables gridx y gridy. En vez de estas variables, se utilizan las constantes **RELATIVE** y **REMAINDER** de GridBagConstraints. RELATIVE especifica que el penúltimo componente de cierta fila deberá colocarse a la derecha del componente anterior en esa fila. REMAINDER especifica que un componente es el último en una fila. Cualquier componente que no sea el penúltimo o el último en una fila, deberá especificar valores para las variables gridwidth y gridheight de GridbagConstraints. La aplicación de las figuras 22.23 y 22.24 ordena los componentes de un esquema Grid-BagLayout, utilizando estas constantes.

En las líneas 21 y 22 se crea un objeto GridBagLayout y se utiliza para establecer el administrador de esquemas del objeto JFrame. Los componentes que se colocan en el esquema GridBagLayout se crean en las líneas 27 a 38; son un objeto JComboBox, un objeto JTextField, un objeto JList y cinco objetos JButton.

El objeto JTextField se agrega primero (líneas 41 a 45). Los valores de weightx y weighty se establecen en 1. La variable fill se establece en BOTH. En la línea 44 se especifica que el objeto JTextField es el último componente de la línea. El objeto JTextField se agrega al panel de contenido mediante una llamada a nuestro método utilitario agregarComponente (declarado en las líneas 79 a 83). El método agregarComponente recibe un argumento Component y utiliza el método setConstraints de GridBagLayout para establecer las restricciones de este objeto Component. El método add adjunta el componente al panel de contenido.

```
 1  // Fig. 22.23: MarcoGridBag2.java
 2  // Demostración de las constantes de GridBagLayout.
 3  import java.awt.GridBagLayout;
 4  import java.awt.GridBagConstraints;
 5  import java.awt.Component;
 6  import javax.swing.JFrame;
 7  import javax.swing.JComboBox;
 8  import javax.swing.JTextField;
 9  import javax.swing.JList;
10  import javax.swing.JButton;
11
12  public class MarcoGridBag2 extends JFrame
13  {
14     private GridBagLayout esquema; // esquema de este marco
15     private GridBagConstraints restricciones; // restricciones de este esquema
16
17     // establece la GUI
18     public MarcoGridBag2()
19     {
20        super( "GridBagLayout" );
21        esquema = new GridBagLayout();
22        setLayout( esquema ); // establece el esquema del marco
23        restricciones = new GridBagConstraints(); // instancia las restricciones
24
25        // crea los componentes de la GUI
26        String metales[] = { "Cobre", "Aluminio", "Plata" };
```

Figura 22.23 | Las constantes RELATIVE y REMAINDER de GridBagConstraints. (Parte 1 de 2).

```
      JComboBox cuadroComb = new JComboBox( metales );

      JTextField campoTexto = new JTextField( "CampoTexto" );

      String fuentes[] = { "Serif", "Monospaced" };
      JList lista = new JList( fuentes );

      String nombres[] = { "cero", "uno", "dos", "tres", "cuatro" };
      JButton botones[] = new JButton[ nombres.length ];

      for ( int cuenta = 0; cuenta < botones.length; cuenta++ )
         botones[ cuenta ] = new JButton( nombres[ cuenta ] );

      // define las restricciones para el componente de la GUI campoTexto
      restricciones.weightx = 1;
      restricciones.weighty = 1;
      restricciones.fill = GridBagConstraints.BOTH;
      restricciones.gridwidth = GridBagConstraints.REMAINDER;
      agregarComponente( campoTexto );

      // botones[0] -- weightx y weighty son 1: fill es BOTH
      restricciones.gridwidth = 1;
      agregarComponente( botones[ 0 ] );

      // botones[1] -- weightx y weighty son 1: fill es BOTH
      restricciones.gridwidth = GridBagConstraints.RELATIVE;
      agregarComponente( botones[ 1 ] );

      // botones[2] -- weightx y weighty son 1: fill es BOTH
      restricciones.gridwidth = GridBagConstraints.REMAINDER;
      agregarComponente( botones[ 2 ] );

      // cuadroComb -- weightx es 1: fill es BOTH
      restricciones.weighty = 0;
      restricciones.gridwidth = GridBagConstraints.REMAINDER;
      agregarComponente( cuadroComb );

      // botones[3] -- weightx es 1: fill es BOTH
      restricciones.weighty = 1;
      restricciones.gridwidth = GridBagConstraints.REMAINDER;
      agregarComponente( botones[ 3 ] );

      // botones[4] -- weightx y weighty son 1: fill es BOTH
      restricciones.gridwidth = GridBagConstraints.RELATIVE;
      agregarComponente( botones[ 4 ] );

      // lista -- weightx y weighty son 1: fill es BOTH
      restricciones.gridwidth = GridBagConstraints.REMAINDER;
      agregarComponente( lista );
   } // fin del constructor de MarcoGridBag2

   // agrega un componente al contenedor
   private void agregarComponente( Component componente )
   {
      esquema.setConstraints( componente, restricciones );
      add( componente ); // agrega el componente
   } // fin del método agregarComponente
} // fin de la clase MarcoGridBag2
```

Figura 22.23 | Las constantes `RELATIVE` y `REMAINDER` de `GridBagConstraints`. (Parte 2 de 2).

El objeto `JButton` `botones[ 0 ]` (líneas 48 y 49) tiene valores de `weightx` y `weighty` de 1. La variable `fill` tiene el valor de `BOTH`. Como `botones[ 0 ]` no es uno de los últimos componentes de la fila, recibe un valor de `gridwidth` de 1, de manera que ocupe solamente una columna. El objeto `JButton` se agrega al panel de contenido mediante una llamada al método utilitario `agregarComponente`.

El objeto `JButton` `botones[ 1 ]` (líneas 52 y 53) tiene valores de `weightx` y `weighty` de 1. La variable `fill` tiene el valor de `BOTH`. En la línea 52 se especifica que el objeto `JButton` se va a colocar de manera relativa al componente anterior. El objeto `JButton` se agrega al objeto `JFrame` mediante una llamada a `agregarComponente`.

El objeto `JButton` `botones[ 2 ]` (líneas 56 y 57) tiene valores de `weightx` y `weighty` de 1. La variable `fill` tiene el valor de `BOTH`. Este objeto `JButton` es el último componente de la línea, por lo que se utiliza `REMAINDER`. El objeto `JButton` se agrega al panel de contenido mediante una llamada a `agregarComponente`.

El objeto `JComboBox` (líneas 60 a 62) tiene un valor de `weightx` de 1 y un valor de `weighty` de 0. El objeto `JComboBox` no crecerá en dirección vertical. Este objeto `JComboBox` es el único componente de la línea, por lo que se utiliza `REMAINDER`. El objeto `JComboBox` se agrega al panel de contenido mediante una llamada a `agregarComponente`.

El objeto `JButton` `botones[ 3 ]` (líneas 65 a 67) tiene valores de `weightx` y `weighty` de 1. La variable `fill` tiene el valor de `BOTH`. Este objeto `JButton` es el único componente de la línea, por lo que se utiliza `REMAINDER`. El objeto `JButton` se agrega al panel de contenido mediante una llamada a `agregarComponente`.

```java
// Fig. 22.24: DemoGridBag2.java
// Demostración de las constantes de GridBagLayout.
import javax.swing.JFrame;

public class DemoGridBag2
{
   public static void main( String args[] )
   {
      MarcoGridBag2 marcoGridBag2 = new MarcoGridBag2();
      marcoGridBag2.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
      marcoGridBag2.setSize( 300, 200 ); // establece el tamaño del marco
      marcoGridBag2.setVisible( true ); // muestra el marco
   } // fin de main
} // fin de la clase DemoGridBag2
```





Figura 22.24 | Clase de prueba para `DemoGridBag2`.

El objeto `JButton` `botones[ 4 ]` (líneas 70 y 71) tiene valores de `weightx` y `weighty` de 1. La variable `fill` tiene el valor de `BOTH`. Este objeto `JButton` es el penúltimo componente de la línea, por lo que se utiliza `RELATIVE`. El objeto `JButton` se agrega al panel de contenido mediante una llamada a `agregarComponente`.

El objeto `JList` (líneas 74 y 75) tiene valores de `weightx` y `weighty` de 1. La variable `fill` tiene el valor de `BOTH`. El objeto `JList` se agrega al panel de contenido mediante una llamada a `agregarComponente`.

22.10 Conclusión

Este capítulo completa nuestra introducción a la GUI. Aquí aprendió acerca de temas más avanzados sobre la GUI, como los menús, controles deslizables, menús contextuales y la interfaz de múltiples documentos. Todos estos componentes se pueden agregar a las aplicaciones existentes, para que sean más fáciles de usar y de entender. En el siguiente capítulo aprenderá acerca del subprocesamiento múltiple, una poderosa herramienta que permite a las aplicaciones utilizar subprocesos para realizar varias tareas a la vez.

Resumen

Sección 22.2 `JSlider`

- Los objetos `JSlider` permiten al usuario seleccionar de entre un rango de valores enteros. Los objetos `JSlider` pueden mostrar marcas más distanciadas, marcas menos distanciadas y etiquetas para las marcas. También soportan el ajuste a la marca, en el que al colocar el indicador entre dos marcas, éste se ajusta a la marca más cercana.
- Si un objeto `JSlider` tiene el foco, la tecla de flecha izquierda y la tecla de flecha derecha hacen que el indicador del objeto `JSlider` se decremente o se incremente en 1. La tecla de flecha hacia abajo y la tecla de flecha hacia arriba también hacen que el indicador del objeto `JSlider` se decremente o incremente en 1, respectivamente. Las teclas *Av Pág* (avance de página) y *Re Pág* (retroceso de página) hacen que el indicador del objeto `JSlider` se decremente o incremente en incrementos de bloque de una décima parte del rango de valores, respectivamente. La *tecla Inicio* desplaza el indicador hacia el valor mínimo del objeto `JSlider`, y la *tecla Fin* desplaza el indicador hacia el valor máximo del objeto `JSlider`.
- El método `setMajorTickSpacing` de la clase `JSlider` establece el espaciado para las marcas en un objeto `JSlider`. El método `setPaintTicks` con un argumento `true` indica que las marcas deben mostrarse.
- Los objetos `JSlider` generan eventos `ChangeEvent` en respuesta a las interacciones del usuario. Un objeto `ChangeListener` declara el método `stateChanged`, el cual puede responder a los eventos `ChangeEvent`.

Sección 22.3 Ventanas: observaciones adicionales

- Todas las ventanas generan eventos de ventana cuando el usuario las manipula. La interfaz `WindowListener` proporciona siete métodos para manejo de eventos de ventana: `windowActivated`, `windowClosed`, `windowClosing`, `windowDeactivated`, `windowDeiconified`, `windowIconified` y `windowOpened`.
- Los menús son una parte integral de las GUIs, ya que permiten al usuario realizar acciones sin atestar innecesariamente una interfaz gráfica de usuario con componentes de la GUI adicionales. En las GUIs de Swing, los menús sólo se pueden adjuntar a objetos de clases que contengan el método `setJMenuBar` (por ejemplo, `JFrame` y `JApplet`).

Sección 22.4 Uso de menús con marcos

- Las clases que se utilizan para declarar menús son `JMenuBar`, `JMenuItem`, `JMenu`, `JCheckBoxMenuItem` y `JRadioButtonMenuItem`.
- Un objeto `JMenuBar` es un contenedor de menús. Un objeto `JMenuItem` es un componente de GUI dentro de un menú que, al ser seleccionado, hace que se realice una acción. Un objeto `JMenu` contiene elementos de menú y puede agregarse a un objeto `JMenuBar` o a otros objetos `JMenu` como submenús.
- Cuando hacemos clic en un menú, éste se expande para mostrar su lista de elementos. El método `addSeparator` de `JMenu` agrega una línea separadora a un menú.
- Al seleccionar un objeto `JCheckBoxMenuItem`, aparece una marca de verificación a la izquierda del elemento de menú. Cuando se selecciona de nuevo el objeto `JCheckBoxMenuItem`, la marca se elimina.
- Cuando se mantienen varios objetos `JRadioButtonMenu` como parte de un objeto `ButtonGroup`, sólo un objeto en el grupo puede seleccionarse en un momento dado. Cuando se selecciona un elemento, aparece un círculo relleno a su izquierda. Cuando se selecciona otro objeto `JRadioButtonMenu`, se elimina el círculo relleno a la izquierda del elemento antes seleccionado.
- El método `setMnemonic` de `AbstractButton` especifica el nemónico para un objeto `AbstractButton`. Los caracteres nemónicos se muestran generalmente subrayados.
- Un cuadro de diálogo modal no permite el acceso a ninguna otra ventana en la aplicación, a menos que se cierre. Los cuadros de diálogo que se muestran con la clase `JOptionPane` son modales. Puede utilizar la clase `JDialog` para crear sus propios cuadros de diálogo modales o no modales.

Sección 22.5 **JPopupMenu**

- Los menús contextuales sensibles al contexto se crean mediante la clase `JPopupMenu`. En la mayoría de los sistemas, el evento de desencadenamiento del menú contextual ocurre cuando el usuario oprime y suelta el botón derecho del ratón. El método `isPopupTrigger` de `MouseEvent` devuelve `true` si ocurrió el evento de desencadenamiento del menú contextual.
- El método `show` de `JPopupMenu` muestra un objeto `JPopupMenu`. El primer argumento especifica el componente de origen, el cual ayuda a determinar en dónde aparecerá el objeto `JPopupMenu`. Los últimos dos argumentos son las coordenadas a partir de la esquina superior izquierda del componente de origen, en donde aparece el objeto `JPopupMenu`.

Sección 22.6 Apariencia visual adaptable

- La clase `UIManager` contiene la clase anidada `LookAndFeelInfo` que mantiene la información acerca de una apariencia visual.
- El método `static getInstalledLookAndFeels` de `UIManager` obtiene un arreglo de objetos `UIManager.LookAndFeelInfo` que describe cada una de las apariencias visuales disponibles.
- El método `static setLookAndFeel` de `UIManager` cambia la apariencia visual. El método `static updateComponentTreeUI` de la clase `SwingUtilities` cambia la apariencia visual de todos los componentes adjuntos a su argumento `Component` a la nueva apariencia visual.

Sección 22.7 **JDesktopPane** *y* **JInternalFrame**

- Muchas de las aplicaciones de la actualidad utilizan una interfaz de múltiples documentos (MDI) para administrar varios documentos abiertos, los cuales se procesan en paralelo. Las clases `JDesktopPane` y `JInternalFrame` de Swing proporcionan soporte para crear interfaces de múltiples documentos.

Sección 22.8 **JTabbedPane**

- Un objeto `JTabbedPane` ordena los componentes de la GUI en capas, de las cuales sólo una puede verse en un momento dado. Los usuarios acceden a cada capa a través de una ficha; muy parecido a las carpetas en un archivero. Cuando el usuario hace clic en una ficha, se muestra la capa apropiada.

Sección 22.9 Administradores de esquemas: **BoxLayout** *y* **GridBagLayout**

- `BoxLayout` es un administrador de esquemas que permite ordenar los componentes de la GUI de izquierda a derecha, o de arriba hacia abajo, en un contenedor.
- La clase `Box` declara un contenedor con `BoxLayout` como su administrador de esquemas predeterminado, y proporciona métodos estáticos para crear un objeto `Box` con un esquema `BoxLayout` horizontal o vertical.
- `GridBagLayout` es un administrador de esquemas similar a `GridLayout`. A diferencia de `GridLayout`, el tamaño de cada componente puede variar y pueden agregarse componentes en cualquier orden.
- Un objeto `GridBagConstraints` describe cómo colocar un componente en un esquema `GridBagLayout`. El método `setConstraints` de la clase `GridBagLayout` recibe un argumento `Component` y un argumento `GridBagConstraints`, y establece las restricciones del objeto `Component`.

Terminología

`add`, método de la clase `JMenuBar`
`addWindowListener`, método de la clase `Window`
ajuste a la marca para `JSlider`
`anchor`, campo de la clase `GridBagConstraints`
apariencia visual adaptable (PLAF)
apariencia visual metálica
área rígida
barra de menús
barra de título
borde
`BOTH`, constante de la clase `GridBagConstraints`
`Box`, clase
`BoxLayout`, clase
`CENTER`, constante de la clase `GridBagConstraints`
`ChangeEvent`, clase
`ChangeListener`, interfaz
componente de origen
`createGlue`, método de la clase `Box`
`createHorizontalBox`, método de la clase `Box`
`createHorizontalGlue`, método de la clase `Box`
`createHorizontalStrut`, método de la clase `Box`
`createRigidArea`, método de la clase `Box`
`createVerticalBox`, método de la clase `Box`
`createVerticalGlue`, método de la clase `Box`
`createVerticalStrut`, método de la clase `Box`
cuadro de diálogo modal
`Dimension`, clase
`dispose`, método de la clase `Window`
documento
`EAST`, constante de la clase `GridBagConstraints`

elemento de menú
envoltura de línea
evento de desencadenamiento de un menú contextual
eventos de ventana
getClassName, método de la clase UIManager.LookAndFeelInfo
getInstalledLookAndFeels, método de la clase UIManager
getPreferredSize, método de la clase Component
getSelectedText, método de la clase JTextComponent
getValue, método de la clase JSlider
GridBagConstraints, clase
GridBagLayout, clase
gridheight, campo de la clase GridBagConstraints
gridwidth, campo de la clase GridBagConstraints
gridx, campo de la clase GridBagConstraints
gridy, campo de la clase GridBagConstraints
HORIZONTAL, constante de GridBagConstraints
indicador de JSlider
interfaz de múltiples documentos (MDI)
isPopupTrigger, método de la clase MouseEvent
isSelected, método de la clase AbstractButton
JCheckBoxMenuItem, clase
JDesktopPane, clase
JDialog, clase
JFrame, clase
JInternalFrame, clase
JMenu, clase
JMenuBar, clase
JMenuItem, clase
JRadioButtonMenuItem, clase
JSlider, clase
JTabbedPane, clase
línea separadora en un menú
LookAndFeelInfo, clase anidada de la clase UIManager
marcas en JSlider
marcas más distanciadas
marcas menos distanciadas de JSlider
menú
menú contextual sensible al contexto
montante vertical
nemónico
NONE, constante de la clase GridBagConstraints
NORTH, constante de la clase GridBagConstraints
NORTHEAST, constante de la clase GridBagConstraints
NORTHWEST, constante de la clase GridBagConstraints
opaco
pack, método de la clase Window
paintComponent, método de la clase JComponent
pegamento horizontal
políticas de desplazamiento
RELATIVE, constante de la clase GridBagConstraints
REMAINDER, constante de la clase GridBagConstraints
setConstraints, método de la clase GridBagLayout
setDefaultCloseOperation, método de la clase JFrame
setInverted, método de la clase JSlider
setJMenuBar, método de la clase JFrame
setLocation, método de la clase Component
setLookAndFeel, método de la clase UIManager
setMajorTickSpacing, método de la clase JSlider
setMnemonic, método de la clase AbstractButton
setOpaque, método de la clase JComponent
setPaintTicks, método de la clase JSlider
setSelected, método de la clase AbstractButton
setVerticalScrollBarPolicy, método de JSlider
show, método de la clase JPopupMenu
SOUTH, constante de la clase GridBagConstraints
SOUTHEAST, constante de la clase GridBagConstraints
SOUTHWEST, constante de la clase GridBagConstraints
stateChanged, método de la interfaz ChangeListener
submenú
SwingUtilities, clase
tecla de acceso rápido
transparencia de un objeto JComponent
UIManager, clase
updateComponentTreeUI, método de la clase SwingUtilities
ventana hija
ventana padre
ventana padre para un cuadro de diálogo
VERTICAL, constante de la clase GridBagConstraints
weightx, campo de la clase GridBagConstraints
weighty, campo de la clase GridBagConstraints
WEST, constante de la clase GridBagConstraints
windowActivated, método de la interfaz WindowListener
windowClosing, método de la interfaz WindowListener
WindowConstants, interfaz
windowDeactivated, método de la interfaz WindowListener
windowDeiconified, método de la interfaz WindowListener
windowIconified, método de la interfaz WindowListener
WindowListener, interfaz
windowOpened, método de la interfaz WindowListener
X_AXIS, constante de la clase Box
Y_AXIS, constante de la clase Box

Ejercicios de autoevaluación

22.1 Complete las siguientes oraciones:

a) La clase ____________ se utiliza para crear un objeto menú.
b) El método ____________ de la clase JMenu coloca una barra separadora en un menú.

c) Los eventos de `JSlider` son manejados por el método ________________ de la interfaz ____________.
d) La variable de instancia ________________ de `GridBagConstraints` se establece en `CENTER` de manera predeterminada.

22.2 Conteste con *verdadero* o *falso* a cada una de las siguientes proposiciones; en caso de ser *falso*, explique por qué.
a) Cuando el programador crea un objeto `JFrame`, como mínimo debe crearse y agregarse un menú al objeto `JFrame`.
b) La variable `fill` pertenece a la clase `GridBagLayout`.
c) La acción de dibujar en un componente de la GUI se realiza con respecto a la coordenada (0, 0) de la esquina superior izquierda del componente.
d) El esquema predeterminado para un objeto `Box` es `BoxLayout`.

22.3 Encuentre el (los) error(es) en cada una de las siguientes instrucciones y explique cómo corregirlo(s).
a) `JMenubar b;`
b) `miSlider = JSlider( 1000, 222, 100, 450 );`
c) `gbc.fill = GridBagConstraints.NORTHWEST; // establece fill`
d)
```
// sobrescribe a paint en un componente de Swing personalizado
public void paintcomponent( Graphics g )
{
   g.drawString( "HOLA", 50, 50 );
} // fin del metodo Saintcomponent
```
e)
```
// crea un objeto JFrame y lo muestra
JFrame f = new JFrame( "Una ventana" );
f.setVisible( true );
```

Respuestas a los ejercicios de autoevaluación

22.1 a) `JMenu`. b) `addSeparator`. c) `stateChanged`, `ChangeListener`. d) `anchor`.

22.2 a) Falso. Un objeto `JFrame` no requiere menús.
b) Falso. La variable `fill` pertenece a la clase `GridBagConstraints`.
c) Verdadero.
d) Verdadero.

22.3 a) `JMenubar` debe ser `JMenuBar`.
b) El primer argumento para el constructor debe ser `SwingConstants.HORIZONTAL` o `SwingConstants.VERTICAL`, y debe utilizarse la palabra clave `new` después del operador `=`.
c) La constante debe ser `BOTH`, `HORIZONTAL`, `VERTICAL` o `NONE`.
d) `paintcomponent` debe ser `paintComponent`, y el método debe llamar a `super.paintComponent( g )` como su primera instrucción.
e) El método `setSize` de `JFrame` debe ser llamado también, para establecer el tamaño de la ventana.

Ejercicios

22.4 Complete las siguientes oraciones:
a) Un objeto `JMenuItem` que es un `JMenu` se llama ________________.
b) El método ________________ adjunta un objeto `JMenuBar` a un objeto `JFrame`.
c) La clase contenedora ________________ tiene un esquema `BoxLayout` predeterminado.
d) Un ________________ administra a un conjunto de ventanas hijas declaradas con la clase `JInternalFrame`.

22.5 Conteste con *verdadero* o *falso* a cada una de las siguientes proposiciones; en caso de ser *falso*, explique por qué.
a) Los menús requieren un objeto `JMenuBar` para poder adjuntarse a un objeto `JFrame`.
b) `BoxLayout` es el administrador de esquemas predeterminado para un objeto `JFrame`.
c) El método `setEditable` es un método de `JTextComponent`.
d) La clase `JFrame` extiende directamente a la clase `Container`.
e) Los objetos `JApplet` pueden contener menús.

22.6 Encuentre el (los) error(es) en cada una de las siguientes instrucciones. Explique cómo corregir el (los) error(es).

a) `x.add( new JMenuItem( "Submenu Color" ) ); // crear submenú`

b) `contenedor.setLayout( m = new GridbagLayout() );`

22.7 Escriba un programa que muestre un círculo de tamaño aleatorio, calcule y muestre su área, radio, diámetro y circunferencia. Use las siguientes ecuaciones: *diametro* = *2* ∞ *radio*, *area* = π ∞ *radio*^{2}, *circunferencia* = *2* ∞ π ∞ *radio*. Use la constante `Math.PI` para pi (π). Todos los dibujos deberán realizarse en una subclase de `JPanel`, y los resultados de los cálculos deberán mostrarse en un objeto `JTextArea` de sólo lectura.

22.8 Mejore el programa del ejercicio 22.7 al permitir al usuario alterar el radio con un objeto `JSlider`. El programa deberá funcionar para todos los radios en el rango de 100 a 200. A medida que cambie el radio, el diámetro, área y circunferencias deberán actualizarse y mostrarse. El radio inicial debe ser 150. Use las ecuaciones del ejercicio 22.7. Todos los dibujos deberán realizarse en una subclase de `JPanel`, y los resultados de los cálculos deberán mostrarse en un objeto `JTextArea` de sólo lectura.

22.9 Explore los efectos de variar los valores de `weightx` y `weighty` del programa de la figura 22.21. ¿Qué ocurre cuando una ranura tiene valores distintos de cero para `weightx` y `weighty`, pero no se le permite rellenar toda el área (es decir, que el valor `fill` no sea `BOTH`)?

22.10 Escriba un programa que utilice el método `paintComponent` para dibujar el valor actual de un objeto `JSlider` en una subclase de `JPanel`. Además, proporcione un objeto `JTextField` en donde pueda introducirse un valor específico. El objeto `JTextField` deberá mostrar el valor actual del objeto `JSlider` en todo momento. Debe usarse un objeto `JLabel` para identificar al objeto `JTextField`. Deben utilizarse los métodos `setValue` y `getValue` de `JSlider`. [*Nota:* El método `setValue` es un método `public` que no devuelve un valor y toma un argumento entero: el valor de `JSlider`, que determina la posición del indicador].

22.11 Modifique el programa de la figura 22.13, agregando un mínimo de dos fichas nuevas.

22.12 Declare a una subclase de `JPanel` llamada `MiSelectorDeColor`, que proporcione tres objetos `JSlider` y tres objetos `JTextField`. Cada objeto `JSlider` representa los valores de 0 a 255 para las partes rojo, azul y verde de un color. Use estos valores como argumentos para el constructor de `Color`, para crear un nuevo objeto `Color`. Muestre el valor actual de cada objeto `JSlider` en el objeto `JTextField` correspondiente. Cuando el usuario cambie el valor del objeto `JSlider`, el objeto `JTextField` deberá cambiar de manera acorde. Use su nuevo componente de la GUI como parte de una aplicación que muestre el valor actual de `Color`, dibujando un rectángulo relleno.

22.13 Modifique la clase `MiSelectorDeColor` del ejercicio 22.12 para permitir al usuario introducir un valor entero en un objeto `JTextField`, para establecer el valor rojo, verde o azul. Cuando el usuario oprima *Intro* en el objeto `JTextField`, deberá establecerse el objeto `JSlider` correspondiente al valor apropiado.

22.14 Modifique la aplicación del ejercicio 22.13 para dibujar el color actual como un rectángulo en una instancia de una subclase de `JPanel`, que proporcione su propio método `paintComponent` para dibujar el rectángulo, y proporcione métodos *establecer* para establecer los valores de rojo, verde y azul para el color actual. Cuando se invoque a uno de los métodos *establecer*, el objeto deberá repintarse automáticamente a sí mismo (mediante `repaint`).

22.15 Modifique la aplicación del ejercicio 22.14 para permitir al usuario arrastrar el ratón a lo largo del panel de dibujo (una subclase de `JPanel`), para dibujar una figura con el color actual. Permita al usuario seleccionar la figura a dibujar.

22.16 Modifique el programa del ejercicio 22.15 para proporcionar al usuario la habilidad de terminar la aplicación, haciendo clic en el cuadro para cerrar en la ventana que se muestre en pantalla, y seleccionando `Salir` de un menú `Archivo`. Use las técnicas que se muestran en la figura 22.5.

22.17 *(Aplicación de dibujo completa)* Utilizando las técnicas desarrolladas en este capítulo y en el capítulo 11, cree una aplicación de dibujo completa. El programa debe utilizar los componentes de la GUI de los capítulos 11 y 22 para permitir al usuario seleccionar la figura, color y características de relleno. Cada figura deberá guardarse en un arreglo de objetos `MiFigura`, en donde `MiFigura` será superclase en su jerarquía de clases de figuras. Use un objeto `JDesktopPane` y objetos `JInternalFrame` para permitir al usuario crear varios dibujos separados, en ventanas hijas separadas. Cree la interfaz de usuario como una ventana hija separada que contenga todos los componentes de la GUI que permitan al usuario determinar las características de la figura a dibujar. El usuario podrá entonces hacer clic en cualquier objeto `JInternalFrame` para dibujar la figura.

23



La definición más general de la belleza… múltiple deidad en la unidad.

—Samuel Taylor Coleridge

No bloquees el camino de la investigación.

—Charles Sanders Peirce

Una persona con un reloj sabe qué hora es;
una persona con dos relojes nunca estará segura.

—Proverbio

Aprenda a laborar y esperar.

—Henry Wadsworth Longfellow

El mundo avanza tan rápido estos días que el hombre que dice que no puede hacer algo, por lo general, se ve interrumpido por alguien que lo está haciendo.

—Elbert Hubbard

Subprocesamiento múltiple

OBJETIVOS

En este capítulo aprenderá a:

- Conocer qué son los subprocesos y por qué son útiles.
- Conocer cómo nos permiten los subprocesos administrar actividades concurrentes.
- Conocer el ciclo de vida de un subproceso.
- Conocer acerca de las prioridades y la programación de subprocesos.
- Crear y ejecutar objetos `Runnable`.
- Conocer acerca de la sincronización de subprocesos.
- Saber qué son las relaciones productor/consumidor y cómo se implementan con el subprocesamiento múltiple.
- Habilitar varios subprocesos para actualizar los componentes de GUI de Swing en forma segura para los subprocesos.
- Conocer acerca de las interfaces `Callable` y `Future`, que podemos usar para ejecutar tareas que devuelven resultados.

Plan general

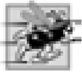
23.1 Introducción

Sería bueno si pudiéramos concentrar nuestra atención en realizar una acción a la vez, y realizarla bien, pero por lo general, eso es difícil. El cuerpo humano realiza una gran variedad de operaciones en **paralelo**; o, como diremos en este capítulo, **operaciones concurrentes**. Por ejemplo, la respiración, la circulación de la sangre, la digestión, la acción de pensar y caminar pueden ocurrir de manera concurrente. Todos los sentidos (vista, tacto, olfato, gusto y oído) pueden emplearse al mismo tiempo. Las computadoras también pueden realizar operaciones concurrentes. Para las computadoras personales es tarea común compilar un programa, enviar un archivo a una impresora y recibir mensajes de correo electrónico a través de una red, de manera concurrente. Sólo las computadoras que tienen múltiples procesadores pueden realmente ejecutar varias instrucciones en forma concurrente. Los sistemas operativos en las computadoras con un solo procesador crean la ilusión de la ejecución concurrente, al alternar rápidamente entre una tarea y otra, pero en dichas computadoras sólo se puede ejecutar una instrucción a la vez.

La mayoría de los lenguajes de programación no permiten a los programadores especificar actividades concurrentes. Por lo general, los lenguajes proporcionan instrucciones de control secuenciales, las cuales permiten a los programadores especificar que sólo debe realizarse una acción a la vez, en donde la ejecución procede a la siguiente acción una vez que la anterior haya terminado. Históricamente, la concurrencia se ha implementado en forma de funciones primitivas del sistema operativo, disponibles sólo para los programadores de sistemas altamente experimentados.

El lenguaje de programación Ada, desarrollado por el Departamento de defensa de los Estados Unidos, hizo que las primitivas de concurrencia estuvieran disponibles ampliamente para los contratistas de defensa dedicados a la construcción de sistemas militares de comando y control. Sin embargo, Ada no se ha utilizado de manera general en las universidades o en la industria.

Java pone las primitivas de concurrencia a disposición del programador de aplicaciones, a través del lenguaje y de las APIs. El programador especifica que una aplicación contiene **subprocesos de ejecución** separados, en donde cada subproceso tiene su propia pila de llamadas a métodos y su propio contador, lo cual le permite ejecutarse concurrentemente con otros subprocesos, al mismo tiempo que comparte los recursos a nivel de aplicación (como la memoria) con estos otros subprocesos. Esta capacidad, llamada **subprocesamiento múltiple**, no está disponible en los lenguajes C y C++ básicos, los cuales influenciaron el diseño de Java.



Tip de rendimiento 23.1

Un problema con las aplicaciones de un solo subproceso es que las actividades que llevan mucho tiempo deben completarse antes de que puedan comenzar otras. En una aplicación con subprocesamiento múltiple, los subprocesos pueden distribuirse entre varios procesadores (si hay disponibles), de manera que se realicen varias tareas en forma concurrente, lo cual permite a la aplicación operar con más eficiencia. El subprocesamiento múltiple también puede incrementar el rendimiento en sistemas con un solo procesador que simulan la concurrencia; cuando un subproceso no puede proceder (debido a que, por ejemplo, está esperando el resultado de una operación de E/S), otro puede usar el procesador.

A diferencia de los lenguajes que no tienen capacidades integradas de subprocesamiento múltiple (como C y C++) y que, por lo tanto, deben realizar llamadas no portables a las primitivas de subprocesamiento múltiple del sistema operativo, Java incluye las primitivas de subprocesamiento múltiple como parte del mismo lenguaje y como parte de sus bibliotecas. Esto facilita la manipulación de los subprocesos de una manera portable entre plataformas.

Hablaremos sobre muchas aplicaciones de la **programación concurrente**. Por ejemplo, cuando se descarga un archivo extenso (como una imagen, un clip de audio o video) a través de Internet, tal vez el usuario no quiera esperar hasta que se descargue todo un clip completo para empezar a reproducirlo. Para resolver este problema podemos poner varios subprocesos a trabajar; uno para descargar el clip y otro para reproducirlo. Estas actividades se llevan a cabo concurrentemente. Para evitar que la reproducción del clip tenga interrupciones, **sincronizamos** (coordinamos las acciones de) los subprocesos de manera que el subproceso de reproducción no empiece sino hasta que haya una cantidad suficiente del clip en memoria, como para mantener ocupado al subproceso de reproducción.

La Máquina Virtual de Java (JVM) utiliza subprocesos también. Además de crear subprocesos para ejecutar un programa, la JVM también puede crear subprocesos para llevar a cabo tareas de mantenimiento, como la recolección de basura.

Escribir programas con subprocesamiento múltiple puede ser un proceso complicado. Aunque la mente humana puede realizar funciones de manera concurrente, a las personas se les dificulta saltar de un tren paralelo de pensamiento al otro. Para ver por qué los programas con subprocesamiento múltiple pueden ser difíciles de escribir y comprender, intente realizar el siguiente experimento: abra tres libros en la página 1 y trate de leerlos concurrentemente. Lea unas cuantas palabras del primer libro; después unas cuantas del segundo y por último otras cuantas del tercero. Luego, regrese a leer las siguientes palabras del primer libro, etcétera. Después de este experimento podrá apreciar muchos de los retos que implica el subprocesamiento múltiple: alternar entre los libros, leer brevemente, recordar en dónde se quedó en cada libro, acercar el libro que está leyendo para poder verlo, hacer a un lado los libros que no está leyendo y, entre todo este caos, ¡tratar de comprender el contenido de cada uno!

La programación de aplicaciones concurrentes es una tarea difícil y propensa a errores. Si descubre que debe usar la sincronización en un programa, debe seguir ciertos lineamientos simples. Primero, utilice las clases existentes de la API de Java (como la clase `ArrayBlockingQueue` que veremos en la sección 23.7, Relación productor/consumidor: `ArrayBlockingQueue`) que administren la sincronización por usted. Las clases en la API de Java están escritas por expertos, han sido probadas y depuradas por completo, operan con eficiencia y le ayudan a evitar trampas y obstáculos comunes. En segundo lugar, si descubre que necesita una funcionalidad más personalizada de la que se proporciona en las APIs de Java, debe usar la palabra clave `synchronized` y los métodos `wait`, `notify` y `notifyAll` de `Object` (que veremos en las secciones 23.5 y 23.8). Por último, si requiere herramientas aún más complejas, entonces debe usar las `interfaces Lock` y `Condition` que se presentan en la sección 23.10.

Sólo los programadores avanzados que estén familiarizados con las trampas y obstáculos comunes de la programación concurrente deben utilizar las `interfaces Lock` y `Condition`. En este capítulo explicaremos estos temas por varias razones: proporcionan una base sólida para comprender cómo las aplicaciones concurrentes sincronizan el acceso a la memoria compartida; los conceptos son importantes de comprender, aun si una aplicación no utiliza estas herramientas de manera explícita; y al demostrarle la complejidad involucrada en el uso de estas características de bajo nivel, esperamos dejar en usted la impresión acerca de la importancia del uso de las herramientas de concurrencia preempaquetadas, siempre que sea posible.

23.2 Estados de los subprocesos: ciclo de vida de un subproceso

En cualquier momento dado, se dice que un subproceso se encuentra en uno de varios **estados de subproceso** (los cuales se muestran en el diagrama de estados de UML de la figura 23.1). Varios de los términos en el diagrama se definen en secciones posteriores.

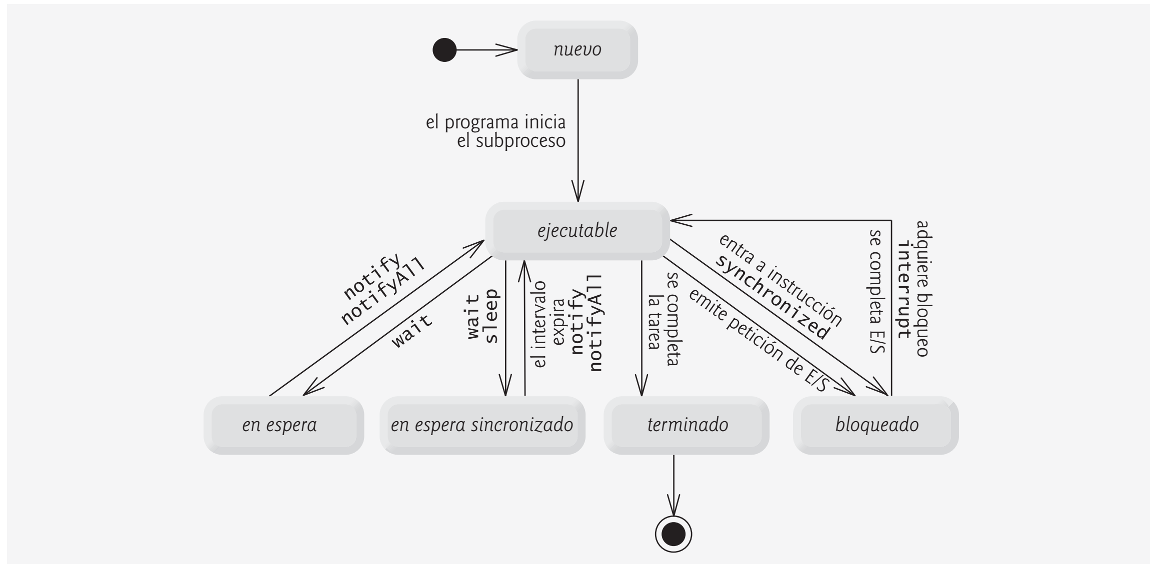


Figura 23.1 | Diagrama de estado de UML del ciclo de vida de un subproceso.

Un nuevo subproceso empieza su ciclo cuando hace la transición al estado ***nuevo***; permanece en este estado hasta que el programa inicia el subproceso, con lo cual se coloca en el estado ***ejecutable***. Se considera que un *subproceso* en el estado ejecutable está ejecutando su tarea.

Algunas veces, un subproceso *ejecutable* cambia al estado ***en espera*** mientras espera a que otro subproceso realice una tarea. Un subproceso *en espera* regresa al estado *ejecutable* sólo cuando otro subproceso notifica al subproceso esperando que puede continuar ejecutándose.

Un subproceso *ejecutable* puede entrar al estado ***en espera sincronizado*** durante un intervalo específico de tiempo. Regresa al estado *ejecutable* cuando ese intervalo de tiempo expira, o cuando ocurre el evento que está esperando. Los subprocesos en *espera sincronizado* y *en espera* no pueden usar un procesador, aun cuando haya uno disponible. Un subproceso *ejecutable* puede cambiar al estado *en espera sincronizado* si proporciona un intervalo de espera opcional cuando está esperando a que otro subproceso realice una tarea. Dicho subproceso regresa al estado *ejecutable* cuando se lo notifica otro subproceso, o cuando expira el intervalo de tiempo; lo que ocurra primero. Otra manera de colocar a un subproceso en el estado *en espera sincronizado* es dejar inactivo un subproceso *ejecutable*. Un **subproceso inactivo** permanece en el estado *en espera sincronizado* durante un periodo designado de tiempo (conocido como **intervalo de inactividad**), después del cual regresa al estado *ejecutable*. Los subprocesos están inactivos cuando, en cierto momento, no tienen una tarea que realizar. Por ejemplo, un procesador de palabras puede contener un subproceso que periódicamente respalde (es decir, escriba una copia de) el documento actual en el disco, para fines de recuperarlo. Si el subproceso no quedara inactivo entre un respaldo y otro, requeriría un ciclo en el que se evaluara continuamente si debe escribir una copia del documento en el disco. Este ciclo consumiría tiempo del procesador sin realizar un trabajo productivo, con lo cual se reduciría el rendimiento del sistema. En este caso, es más eficiente para el subproceso especificar un intervalo de inactividad (equivalente al periodo entre respaldos sucesivos) y entrar al estado *en espera sincronizado*. Este subproceso se regresa al estado *ejecutable* cuando expire su intervalo de inactividad, punto en el cual escribe una copia del documento en el disco y vuelve a entrar al estado *en espera sincronizado*.

Un subproceso *ejecutable* cambia al estado ***bloqueado*** cuando trata de realizar una tarea que no puede completarse inmediatamente, y debe esperar temporalmente hasta que se complete esa tarea. Por ejemplo, cuando un subproceso emite una petición de entrada/salida, el sistema operativo bloquea el subproceso para que no se ejecute sino hasta que se complete la petición de E/S; en ese punto, el subproceso *bloqueado* cambia al estado *ejecutable*, para poder continuar su ejecución. Un subproceso *bloqueado* no puede usar un procesador, aun si hay uno disponible.

Un subproceso ejecutable entra al estado ***terminado*** (algunas veces conocido como el estado ***muerto***) cuando completa exitosamente su tarea, o termina de alguna otra forma (tal vez debido a un error). En el diagrama

de estados de UML de la figura 23.1, el estado *terminado* va seguido por el estado final de UML (el símbolo de viñeta) para indicar el final de las transiciones de estado.

A nivel del sistema operativo, el estado *ejecutable* de Java generalmente abarca dos estados separados (figura 23.2). El sistema operativo oculta estos estados de la Máquina Virtual de Java (JVM), la cual sólo ve el estado *ejecutable*. Cuando un subproceso cambia por primera vez al estado *ejecutable* desde el estado *nuevo*, el subproceso se encuentra en el estado ***listo***. Un subproceso *listo* entra al estado ***en ejecución*** (es decir, empieza su ejecución) cuando el sistema operativo lo asigna a un procesador; a esto también se le conoce como **despachar el subproceso**. En la mayoría de los sistemas operativos, a cada subproceso se le otorga una pequeña cantidad de tiempo del procesador (lo cual se conoce como ***quantum*** o **intervalo de tiempo**) en la que debe realizar su tarea. (El proceso de decidir qué tan grande debe ser el quantum de tiempo es un tema clave en los cursos de sistemas operativos). Cuando expira su quantum, el subproceso regresa al estado *listo* y el sistema operativo asigna otro subproceso al procesador (vea la sección 23.3). Las transiciones entre los estados *listo* y en *ejecución* las maneja únicamente el sistema operativo. La JVM no "ve" las transiciones; simplemente ve el subproceso como *ejecutable* y deja al sistema operativo la decisión de cambiar el subproceso entre *listo* y *ejecutable*. El proceso que utiliza un sistema operativo para determinar qué subproceso debe despachar se conoce como **programación de subprocesos**, y depende de las prioridades de los subprocesos (que veremos en la siguiente sección).

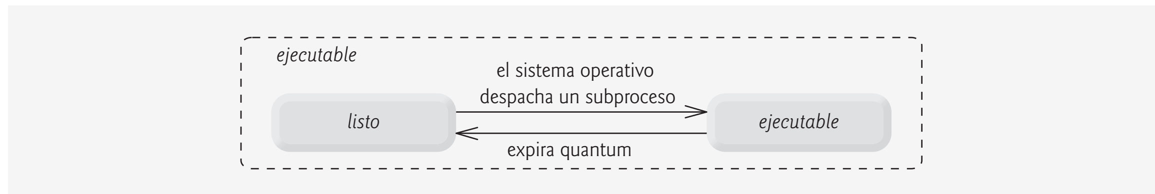


Figura 23.2 | Vista interna del sistema operativo del estado *ejecutable* de Java.

23.3 Prioridades y programación de subprocesos

Todo subproceso en Java tiene una **prioridad de subproceso**, la cual ayuda al sistema operativo a determinar el orden en el que se programan los subprocesos. Las prioridades de Java varían entre `MIN_PRIORITY` (una constante de 1) y `MAX_PRIORITY` (una constante de 10). De manera predeterminada, cada subproceso recibe la prioridad `NORM_PRIORITY` (una constante de 5). Cada nuevo subproceso hereda la prioridad del subproceso que lo creó. De manera informal, los subprocesos de mayor prioridad son más importantes para un programa, y se les debe asignar tiempo del procesador antes que a los subprocesos de menor prioridad. Sin embargo, las prioridades de los subprocesos no garantizan el orden en el que se ejecutan los subprocesos.

[*Nota:* las constantes (`MAX_PRIORITY`, `MIN_PRIORITY` y `NORM_PRIORITY`) se declaran en la clase `Thread`. Se recomienda no crear y usar explícitamente objetos `Thread` para implementar la concurrencia, sino utilizar mejor la interfaz `Executor` (que describiremos en la sección 23.4.2). La clase `Thread` contiene varios métodos `static` útiles, los cuales describiremos más adelante en este capítulo].

La mayoría de los sistemas operativos soportan los intervalos de tiempo (timeslicing), que permiten a los subprocesos de igual prioridad compartir un procesador. Sin el intervalo de tiempo, cada subproceso en un conjunto de subprocesos de igual prioridad se ejecuta hasta completarse (a menos que deje el estado *ejecutable* y entre al estado *en espera* o en *espera sincronizado*, o lo interrumpa un subproceso de mayor prioridad) antes que otros subprocesos de igual prioridad tengan oportunidad de ejecutarse. Con el intervalo de tiempo, aun si un subproceso no ha terminado de ejecutarse cuando expira su quantum, el procesador se quita del subproceso y pasa al siguiente subproceso de igual prioridad, si hay uno disponible.

El **programador de subprocesos** de un sistema operativo determina cuál subproceso se debe ejecutar a continuación. Una implementación simple del programador de subprocesos mantiene el subproceso de mayor prioridad en *ejecución* en todo momento y, si hay más de un subproceso de mayor prioridad, asegura que todos esos subprocesos se ejecuten durante un quantum cada uno, en forma **cíclica (round-robin)**. En la figura 23.3 se muestra una **cola de prioridades multinivel** para los subprocesos. En la figura, suponiendo que hay una computadora con un solo procesador, los subprocesos A y B se ejecutan cada uno durante un quantum, en forma cíclica hasta que ambos subprocesos terminan su ejecución. Esto significa que A obtiene un quantum de tiempo

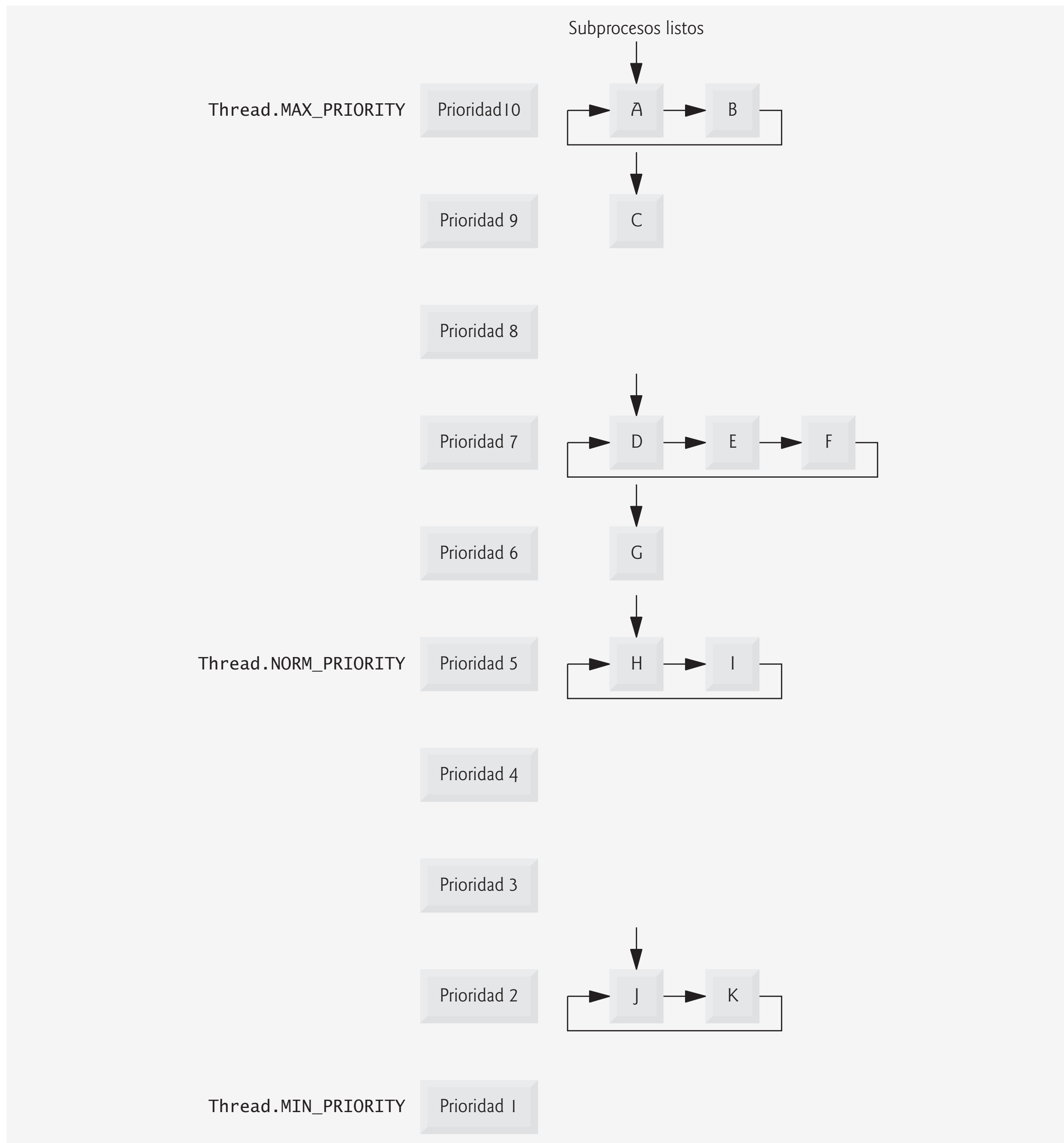


Figura 23.3 | Programación de prioridad de subprocesos.

para ejecutarse. Después B obtiene un quantum. Luego A obtiene otro quantum. Después B obtiene otro. Esto continúa hasta que un subproceso se completa. Entonces, el procesador dedica todo su poder al subproceso restante (a menos que esté listo otro subproceso de prioridad 10). A continuación, el subproceso C se ejecuta hasta completarse (suponiendo que no llegan subprocesos de mayor prioridad). Los subprocesos D, E y F se ejecutan cada uno durante un quantum, en forma cíclica hasta que todos terminan de ejecutarse (de nuevo, suponiendo que no llegan procesos de mayor prioridad). Este proceso continúa hasta que todos los subprocesos se ejecutan hasta completarse.

Cuando un subproceso de mayor prioridad entra al estado *listo*, el sistema operativo generalmente sustituye el subproceso actual en *ejecución* para dar preferencia al subproceso de mayor prioridad (una operación conocida como **programación preferente**). Dependiendo del sistema operativo, los subprocesos de mayor prioridad

podrían posponer (tal vez de manera indefinida) la ejecución de los subprocesos de menor prioridad. Comúnmente, a dicho **aplazamiento indefinido** se le conoce, en forma más colorida, como **inanición**.

Java cuenta con herramientas de concurrencia de alto nivel para ocultar parte de esta complejidad, y hacer que los programas sean menos propensos a errores. Las prioridades de los subprocesos se utilizan en segundo plano para interactuar con el sistema operativo, pero la mayoría de los programadores que utilizan subprocesamiento múltiple en Java no tienen que preocuparse por configurar y ajustar las prioridades de los subprocesos.



Tip de portabilidad 23.1

La programación de subprocesos es independiente de la plataforma; el comportamiento de un programa con subprocesamiento múltiple podría variar entre las distintas implementaciones de Java.



Tip de portabilidad 23.2

Al diseñar programas con subprocesamiento múltiple, considere las capacidades de subprocesamiento de las plataformas en las que se ejecutarán esos programas. Si utiliza prioridades distintas a las predeterminadas, el comportamiento de sus programas será específico para la plataforma en la que se ejecuten. Si su meta es la portabilidad, no ajuste las prioridades de los subprocesos.

23.4 Creación y ejecución de subprocesos

El medio preferido de crear aplicaciones en Java con subprocesamiento múltiple es mediante la implementación de la interfaz `Runnable` (del paquete `java.lang`). Un objeto `Runnable` representa una "tarea" que puede ejecutarse concurrentemente con otras tareas. La interfaz `Runnable` declara un solo método, `run`, el cual contiene el código que define la tarea que debe realizar un objeto `Runnable`. Cuando se crea e inicia un subproceso que ejecuta un objeto `Runnable`, el subproceso llama al método `run` del objeto `Runnable`, el cual se ejecuta en el nuevo subproceso.

23.4.1 Objetos `Runnable` y la clase `Thread`

La clase `TareaImprimir` (figura 23.4) implementa a `Runnable` (línea 5), de manera que puedan ejecutarse varios objetos `TareaImprimir` en forma concurrente. La variable `tiempoInactividad` (línea 7) almacena un valor entero aleatorio de 0 a 5 segundos, que se crea en el constructor de `TareaImprimir` (línea 16). Cada subproceso que ejecuta a un objeto `TareaImprimir` permanece inactivo durante la cantidad de tiempo especificado por `tiempoInactividad`, después imprime el nombre de la tarea y un mensaje indicando que terminó su inactividad.

```
1  // Fig. 23.4: TareaImprimir.java
2  // La clase TareaImprimir permanece inactiva durante un tiempo aleatorio entre 0 y 5
   segundos
3  import java.util.Random;
4
5  public class TareaImprimir implements Runnable
6  {
7     private final int tiempoInactividad; // tiempo de inactividad aleatorio para el
      subproceso
8     private final String nombreTarea; // nombre de la tarea
9     private final static Random generador = new Random();
10
11    public TareaImprimir( String nombre )
12    {
13       nombreTarea = nombre; // establece el nombre de la tarea
14
15       // elige un tiempo de inactividad aleatorio entre 0 y 5 segundos
16       tiempoInactividad = generador.nextInt( 5000 ); // milisegundos
```

Figura 23.4 | La clase `TareaImprimir` permanece inactiva durante un tiempo aleatorio de 0 a 5 segundos. (Parte 1 de 2).

```
17    } // fin del constructor de TareaImprimir
18
19    // el método run contiene el código que ejecutará un subproceso
20    public void run()
21    {
22       try // deja el subproceso inactivo durante tiempoInactividad segundos
23       {
24          System.out.printf( "%s va a estar inactivo durante %d milisegundos.\n",
25             nombreTarea, tiempoInactividad );
26          Thread.sleep( tiempoInactividad ); // deja el subproceso inactivo
27       } // fin de try
28       catch ( InterruptedException excepcion )
29       {
30          System.out.printf( "%s %s\n", nombreTarea,
31             "termino en forma prematura, debido a la interrupcion" );
32       } // fin de catch
33
34       // imprime el nombre de la tarea
35       System.out.printf( "%s termino su inactividad\n", nombreTarea );
36    } // fin del método run
37 } // fin de la clase TareaImprimir
```

Figura 23.4 | La clase `TareaImprimir` permanece inactiva durante un tiempo aleatorio de 0 a 5 segundos. (Parte 2 de 2).

Un objeto `TareaImprimir` se ejecuta cuando un subproceso llama al método `run` de `TareaImprimir`. En las líneas 24 y 25 se muestra un mensaje, indicando el nombre de la tarea actual en ejecución y que ésta va a quedar inactiva durante `tiempoInactividad` milisegundos. En la línea 26 se invoca al método `static sleep` de la clase `Thread`, para colocar el subproceso en el estado *en espera sincronizado* durante la cantidad especificada de tiempo. En este punto, el subproceso pierde al procesador y el sistema permite que otro subproceso se ejecute. Cuando el subproceso despierta, vuelve a entrar al estado *ejecutable*. Cuando el objeto `TareaImprimir` se asigna a otro procesador de nuevo, en la línea 35 se imprime un mensaje indicando que la tarea dejó de estar inactiva, y después el método `run` termina. Observe que la instrucción `catch` en las líneas 28 a 32 es obligatoria, ya que el método `sleep` podría lanzar una excepción **`InterruptedException`** (verificada) si se hace una llamada al método **`interrupt`** del subproceso inactivo.

En la figura 23.5 se crean objetos `Thread` para ejecutar objetos `TareaImprimir`. En las líneas 12 a 14 se crean tres subprocesos, cada uno de los cuales especifica un nuevo objeto `TareaImprimir` a ejecutar. En las líneas 19 a 21 se hace una llamada al método `start` de `Thread` en cada uno de los subprocesos; cada llamada invoca al método `run` del objeto `Runnable` correspondiente para ejecutar la tarea. En la línea 23 se imprime un mensaje indicando que se han iniciado todas las tareas de los subprocesos, y que el método `main` terminó su ejecución.

```
1  // Fig. 23.5: CreadorSubproceso.java
2  // Creación e inicio de tres subprocesos para ejecutar objetos Runnable.
3  import java.lang.Thread;
4
5  public class CreadorSubproceso
6  {
7     public static void main( String[] args )
8     {
9        System.out.println( "Creacion de subprocesos" );
10
11       // crea cada subproceso con un nuevo objeto Runnable
12       Thread subproceso1 = new Thread( new TareaImprimir( "tarea1" ) );
13       Thread subproceso2 = new Thread( new TareaImprimir( "tarea2" ) );
14       Thread subproceso3 = new Thread( new TareaImprimir( "tarea3" ) );
```

Figura 23.5 | Creación e inicio de tres subprocesos para ejecutar objetos `Runnable`. (Parte 1 de 2).

```
15
16        System.out.println( "Subprocesos creados, iniciando tareas." );
17
18        // inicia los subprocesos y los coloca en el estado "en ejecución"
19        subproceso1.start(); // invoca al método run de tarea1
20        subproceso2.start(); // invoca al método run de tarea2
21        subproceso3.start(); // invoca al método run de tarea3
22
23        System.out.println( "Tareas iniciadas, main termina.\n" );
24     } // fin de main
25  } // fin de la clase CreadorSubproceso
```

```
Creacion de subprocesos
Subprocesos creados, iniciando tareas
Tareas iniciadas, main termina

Tarea3 va a estar inactivo durante 491 milisegundos
tarea2 va a estar inactivo durante 71 milisegundos
Tarea1 va a estar inactivo durante 3549 milisegundos
tarea2 termino su inactividad
tarea3 termino su inactividad
tarea1 termino su inactividad
```

```
Creacion de subprocesos
Subprocesos creados, iniciando tareas
tarea1 va a estar inactivo durante 4666 milisegundos
tarea2 va a estar inactivo durante 48 milisegundos
tarea3 va a estar inactivo durante 3924 milisegundos
Tareas iniciadas, main termina

subproceso2 termino su inactividad
subproceso3 termino su inactividad
subproceso1 termino su inactividad
```

Figura 23.5 | Creación e inicio de tres subprocesos para ejecutar objetos `Runnable`. (Parte 2 de 2).

El código en el método `main` se ejecuta en el **subproceso principal**, un subproceso creado por la JVM. El código en el método `run` de `TareaImprimir` (líneas 20 a 36 de la figura 23.4) se ejecuta en los subprocesos creados en las líneas 12 a 14 de la figura 23.5. Cuando termina el método `main`, el programa en sí continúa ejecutándose, ya que aún hay subprocesos vivos (es decir, alguno de los tres subprocesos que creamos no ha llegado aún al estado *terminado*). El programa no terminará sino hasta que su último subproceso termine de ejecutarse, punto en el cual la JVM también terminará.

Los resultados de ejemplo para este programa muestran el nombre de cada tarea y el tiempo de inactividad, al momento en que el subproceso queda inactivo. El subproceso con el tiempo de inactividad más corto es el que normalmente se despierta primero, indicando que acaba de dejar de estar inactivo y termina. En la sección 23.9 hablaremos sobre las cuestiones acerca del subprocesamiento múltiple que podrían evitar que el subproceso con el tiempo de inactividad más corto se despertara primero. En el primer conjunto de resultados, el subproceso principal termina antes de que cualquiera de los otros subprocesos impriman sus nombres y tiempos de inactividad. Esto muestra que el subproceso principal se ejecuta hasta completarse antes que cualquiera de los otros subprocesos tengan la oportunidad de ejecutarse. En el segundo conjunto de resultados, todos los subprocesos imprimen sus nombres y tiempos de inactividad antes de que termine el subproceso principal. Esto muestra que el sistema operativo permitió a los otros subprocesos ejecutarse antes de que terminara el subproceso principal. Éste es un ejemplo de la programación cíclica (round-robin) que vimos en la sección 23.3. Además, observe que en el primer conjunto de resultados de ejemplo, `tarea3` queda inactivo primero y `tarea1` queda inactivo al último, aun cuando empezamos el subproceso de `tarea1` primero. Esto ilustra el hecho de que no podemos predecir el orden en el que se programarán los subprocesos, aun si conocemos el orden en el que se crearon e iniciaron. Por

último, en cada uno de los conjuntos de resultados, observe que el orden en el que los subprocesos indican que dejaron de estar inactivos coincide con los tiempos de inactividad de menor a mayor de los tres subprocesos. Aunque éste es el orden razonable e intuitivo para que estos subprocesos terminen sus tareas, no se garantiza que los subprocesos vayan a terminar en este orden.

23.4.2 Administración de subprocesos con el marco de trabajo Executor

Aunque es posible crear subprocesos en forma explícita, como en la figura 23.5, se recomienda utilizar la interfaz Executor para administrar la ejecución de objetos Runnable de manera automática. Por lo general, un objeto **Executor** crea y administra un grupo de subprocesos, al cual se le denomina **reserva de subprocesos**, para ejecutar objetos Runnable. Usted puede crear sus propios subprocesos, pero el uso de un objeto Executor tiene muchas ventajas. Los objetos Executor pueden reutilizar los subprocesos existentes para eliminar la sobrecarga de crear un nuevo subproceso para cada tarea, y pueden mejorar el rendimiento al optimizar el número de subprocesos, con lo cual se asegura que el procesador se mantenga ocupado, sin crear demasiados subprocesos como para que la aplicación se quede sin recursos.

La interfaz Executor declara un solo método llamado **execute**, el cual acepta un objeto Runnable como argumento. El objeto Executor asigna cada objeto Runnable que se pasa a su método execute a uno de los subprocesos disponibles en la reserva. Si no hay subprocesos disponibles, el objeto Executor crea un nuevo subproceso, o espera a que haya uno disponible y lo asigna al objeto Runnable que se pasó al método execute.

La interfaz **ExecutorService** (del paquete java.util.concurrent) es una interfaz que extiende a Executor y declara varios métodos más para administrar el ciclo de vida de un objeto Executor. Un objeto que implementa a la interfaz ExecutorService se puede crear mediante el uso de los métodos static declarados en la clase Executors (del paquete java.util.concurrent). Utilizaremos la interfaz ExecutorService y un método de la clase **Executors** en la siguiente aplicación, la cual ejecuta tres tareas.

En la figura 23.6 se utiliza un objeto ExecutorService para administrar subprocesos que ejecuten objetos TareaImprimir. El método main (líneas 8 a 29) crea y nombra tres objetos TareaImprimir (líneas 11 a 13). En la línea 18 se utiliza el método **newCachedThreadPool** de Executors para obtener un objeto ExecutorService que crea nuevos subprocesos, según los va necesitando la aplicación. Estos subprocesos los utiliza ejecutorSubprocesos para ejecutar los objetos Runnable.

```
1  // Fig. 23.6: EjecutorTareas.java
2  // Uso de un objeto ExecutorService para ejecutar objetos Runnable.
3  import java.util.concurrent.Executors;
4  import java.util.concurrent.ExecutorService;
5
6  public class EjecutorTareas
7  {
8     public static void main( String[] args )
9     {
10       // crea y nombra cada objeto Runnable
11       TareaImprimir tarea1 = new TareaImprimir( "tarea1" );
12       TareaImprimir tarea2 = new TareaImprimir( "tarea2" );
13       TareaImprimir tarea3 = new TareaImprimir( "tarea3" );
14
15       System.out.println( "Iniciando Executor" );
16
17       // crea objeto ExecutorService para administrar los subprocesos
18       ExecutorService ejecutorSubprocesos = Executors.newCachedThreadPool();
19
20       // inicia los subprocesos y los coloca en el estado ejecutable
21       ejecutorSubprocesos.execute( tarea1 ); // inicia tarea1
22       ejecutorSubprocesos.execute( tarea2 ); // inicia tarea2
23       ejecutorSubprocesos.execute( tarea3 ); // inicia tarea3
24
```

Figura 23.6 | Uso de un objeto ExecutorService para ejecutar objetos Runnable. (Parte 1 de 2).

```
25          // cierra los subprocesos trabajadores cuando terminan sus tareas
26          ejecutorSubprocesos.shutdown();
27
28          System.out.println( "Tareas iniciadas, main termina.\n" );
29       } // fin de main
30    } // fin de la clase EjecutorTareas
```

```
Iniciando Executor
Tareas iniciadas, main termina

tarea1 va a estar inactivo durante 4806 milisegundos
tarea2 va a estar inactivo durante 2513 milisegundos
tarea3 va a estar inactivo durante 1132 milisegundos
tarea3 termino su inactividad
tarea2 termino su inactividad
tarea1 termino su inactividad
```

```
Iniciando Executor
tarea1 va a estar inactivo durante 1342 milisegundos
tarea2 va a estar inactivo durante 277 milisegundos
tarea3 va a estar inactivo durante 2737 milisegundos
Tareas iniciadas, main termina

subproceso2 termino su inactividad
subproceso1 termino su inactividad
subproceso3 termino su inactividad
```

Figura 23.6 | Uso de un objeto `ExecutorService` para ejecutar objetos `Runnable`. (Parte 2 de 2).

En cada una de las líneas 21 a 23 se invoca el método `execute` de `ExecutorService`. Este método ejecuta el objeto `Runnable` que recibe como argumento (en este caso, un objeto `TareaImprimir`) en algún momento en el futuro. La tarea especificada puede ejecutarse en uno de los subprocesos en la reserva de `ExecutorService`, en un nuevo subproceso creado para ejecutarla, o en el subproceso que llamó al método `execute`; el objeto `ExecutorService` administra estos detalles. El método `execute` regresa inmediatamente después de cada invocación; el programa no espera a que termine cada objeto `TareaImprimir`. En la línea 26 se hace una llamada al método **`shutdown`** de `ExecutorService`, el cual notifica al objeto `ExecutorService` para que deje de aceptar nuevas tareas, pero continúa ejecutando las tareas que ya se hayan enviado. Una vez que se han completado todos los objetos `Runnable` enviados anteriormente, el objeto `ejecutorSubprocesos` termina. En la línea 28 se imprime un mensaje indicando que se iniciaron las tareas y que el subproceso principal está terminando su ejecución. Los conjuntos de resultados de ejemplo son similares a los del programa anterior y, de nuevo, demuestran el no determinismo de la programación de subprocesos.

23.5 Sincronización de subprocesos

Cuando varios subprocesos comparten un objeto, y éste puede ser modificado por uno o más de los subprocesos, pueden ocurrir resultados indeterminados (como veremos en los ejemplos), a menos que el acceso al objeto compartido se administre de manera apropiada. Si un subproceso se encuentra en el proceso de actualizar a un objeto compartido, y otro subproceso trata de actualizarlo también, no queda claro cuál actualización del subproceso se lleva a cabo. Cuando esto ocurre, el comportamiento del programa no puede determinarse; algunas veces el programa producirá los resultados correctos; sin embargo, en otras ocasiones producirá los resultados incorrectos. En cualquier caso, no habrá indicación de que el objeto compartido se manipuló en forma incorrecta.

El problema puede resolverse si se da a un subproceso a la vez el *acceso exclusivo* al código que manipula al objeto compartido. Durante ese tiempo, otros subprocesos que deseen manipular el objeto deben mantenerse en espera. Cuando el subproceso con acceso exclusivo al objeto termina de manipularlo, a uno de los subprocesos que estaba en espera se le debe permitir que continúe ejecutándose. Este proceso, conocido como **sincronización de subprocesos**, coordina el acceso a los datos compartidos por varios subprocesos concurrentes. Al sincronizar

los subprocesos de esta forma, podemos asegurar que cada subproceso que accede a un objeto compartido excluye a los demás subprocesos de hacerlo en forma simultánea; a esto se le conoce como **exclusión mutua**.

Una manera de realizar la sincronización es mediante los **monitores** integrados en Java. Cada objeto tiene un monitor y un **bloqueo de monitor** (o **bloqueo intrínseco**). El monitor asegura que el bloqueo de monitor de su objeto se mantenga por un máximo de sólo un subproceso a la vez. Así, los monitores y los bloqueos de monitor se pueden utilizar para imponer la exclusión mutua. Si una operación requiere que el subproceso en ejecución mantenga un bloqueo mientras se realiza la operación, un subproceso debe adquirir el bloqueo para poder continuar con la operación. Otros subprocesos que traten de realizar una operación que requiera el mismo bloqueo permanecerán bloqueados hasta que el primer subproceso libere el *bloqueo*, punto en el cual los subprocesos bloqueados pueden tratar de adquirir el *bloqueo* y continuar con la operación.

Para especificar que un subproceso debe mantener un bloqueo de monitor para ejecutar un bloque de código, el código debe colocarse en una **instrucción `synchronized`**. Se dice que dicho código está **protegido** por el bloqueo de monitor; un subproceso debe **adquirir el bloqueo** para ejecutar las instrucciones `synchronized`. El monitor sólo permite que un subproceso a la vez ejecute instrucciones dentro de bloques `synchronized` que se bloqueen en el mismo objeto, ya que sólo un subproceso a la vez puede mantener el bloqueo de monitor. Las instrucciones `synchronized` se declaran mediante la **palabra clave `synchronized`**:

```
synchronized ( objeto )
{
    Instrucciones
} // fin de la instrucción synchronized
```

en donde objeto es el *objeto* cuyo bloqueo de monitor se va a adquirir; generalmente, *objeto* es `this` si es el objeto en el que aparece la instrucción `synchronized`. Si varias instrucciones `synchronized` están tratando de ejecutarse en un objeto al mismo tiempo, sólo una de ellas puede estar activa en el objeto; todos los demás subprocesos que traten de entrar a una instrucción `synchronized` en el mismo objeto se colocan en el estado *bloqueado*.

Cuando una instrucción `synchronized` termina de ejecutarse, el bloqueo de monitor del objeto se libera y el sistema operativo puede permitir que uno de los subprocesos *bloqueados*, que intentan entrar a una instrucción `synchronized`, adquieran el bloqueo para continuar. Java también permite los **métodos `synchronized`**. Dicho método es equivalente a una instrucción `synchronized` que encierra el cuerpo completo de un método, y que utiliza a `this` como el objeto cuyo bloqueo de monitor se va a adquirir. Puede especificar un método como `synchronized`; para ello, coloque la palabra clave `synchronized` antes del tipo de valor de retorno del método en su declaración.

23.5.1 Cómo compartir datos sin sincronización

Ahora presentaremos un ejemplo para ilustrar los peligros de compartir un objeto entre varios subprocesos sin una sincronización apropiada. En este ejemplo, dos objetos `Runnable` mantienen referencias a un solo arreglo entero. Cada objeto `Runnable` escribe cinco valores en el arreglo, y después termina. Tal vez esto parezca inofensivo, pero puede provocar errores si el arreglo se manipula sin sincronización.

La clase `ArregloSimple`

Un objeto de la clase `ArregloSimple` (figura 23.7) se compartirá entre varios subprocesos. `ArregloSimple` permitirá que esos subprocesos coloquen valores `int` en `arreglo` (declarado en la línea 7). En la línea 8 se inicializa la variable `indiceEscritura`, la cual se utilizará para determinar el elemento del arreglo en el que se debe escribir a continuación. El constructor (líneas 12 a 15) crea un arreglo entero del tamaño deseado.

```
1  // Fig. 23.7: ArregloSimple.java
2  // Clase que administra un arreglo simple para compartirlo entre varios subprocesos.
3  import java.util.Random;
4
5  public class ArregloSimple // PRECAUCIÓN: ¡NO ES SEGURO PARA LOS SUBPROCESOS!
6  {
```

Figura 23.7 | Clase que administra un arreglo entero para compartirlo entre varios subprocesos. (Parte 1 de 2).

```
7     private final int arreglo[]; // el arreglo entero compartido
8     private int indiceEscritura = 0; // índice del siguiente elemento a escribir
9     private final static Random generador = new Random();
10
11    // construye un objeto ArregloSimple de un tamaño dado
12    public ArregloSimple( int tamanio )
13    {
14       arreglo = new int[ tamanio ];
15    } // fin del constructor
16
17    // agrega un valor al arreglo compartido
18    public void agregar( int valor )
19    {
20       int posicion = indiceEscritura; // almacena el índice de escritura
21
22       try
23       {
24          // pone el subproceso en inactividad de 0 a 499 milisegundos
25          Thread.sleep( generador.nextInt( 500 ) );
26       } // fin de try
27       catch ( InterruptedException ex )
28       {
29          ex.printStackTrace();
30       } // fin de catch
31
32       // coloca el valor en el elemento apropiado
33       arreglo[ posicion ] = valor;
34       System.out.printf( "%s escribio %2d en el elemento %d.\n",
35          Thread.currentThread().getName(), valor, posicion );
36
37       ++indiceEscritura; // incrementa el índice del siguiente elemento a escribir
38       System.out.printf( "Siguiente indice de escritura: %d\n", indiceEscritura );
39    } // fin del método agregar
40
41    // se utiliza para imprimir el contenido del arreglo entero compartido
42    public String toString()
43    {
44       String cadenaArreglo = "\nContenido de ArregloSimple:\n";
45
46       for ( int i = 0; i < arreglo.length; i++ )
47          cadenaArreglo += arreglo[ i ] + " ";
48
49       return cadenaArreglo;
50    } // fin del método toString
51 } // fin de la clase ArregloSimple
```

Figura 23.7 | Clase que administra un arreglo entero para compartirlo entre varios subprocesos. (Parte 2 de 2).

El método `agregar` (líneas 18 a 39) permite insertar nuevos valores al final del arreglo. En la línea 20 se almacena el valor de `indiceEscritura` actual. En la línea 25, el subproceso que invoca a `agregar` queda inactivo durante un intervalo aleatorio de 0 a 499 milisegundos. Esto se hace para que los problemas asociados con el acceso desincronizado a los datos compartidos sean más obvios. Una vez que el subproceso deja de estar inactivo, en la línea 33 se inserta el valor que se pasa a agregar en el arreglo, en el elemento especificado por `posicion`. En las líneas 34 y 35 se imprime un mensaje, indicando el nombre del subproceso en ejecución, el valor que se insertó en el arreglo y en dónde se insertó. En la línea 37 se incrementa `indiceEscritura`, de manera que la siguiente llamada a `agregar` insertará un valor en el siguiente elemento del arreglo. En las líneas 42 a 50 se sobrescribe el método `toString` para crear una representación `String` del contenido del arreglo.

La clase **EscritorArreglo**

La clase `EscritorArreglo` (figura 23.8) implementa a la interfaz `Runnable` para definir una tarea para insertar valores en un objeto `ArregloSimple`. El constructor (líneas 10 a 14) recibe dos argumentos: un valor entero, que viene siendo el primer `valor` que insertará esta tarea en el objeto `ArregloSimple`, y una referencia al objeto `ArregloSimple`. En la línea 20 se invoca el método `agregar` en el objeto `ArregloSimple`. La tarea se completa una vez que se agregan tres enteros consecutivos, empezando con `valorInicial`, al objeto `ArregloSimple`.

```
1  // Fig. 23.8: EscritorArreglo.java
2  // Agrega enteros a un arreglo compartido con otros objetos Runnable
3  import java.lang.Runnable;
4
5  public class EscritorArreglo implements Runnable
6  {
7     private final ArregloSimple arregloSimpleCompartido;
8     private final int valorInicial;
9
10    public EscritorArreglo( int valor, ArregloSimple arreglo )
11    {
12       valorInicial = valor;
13       arregloSimpleCompartido= arreglo;
14    } // fin del constructor
15
16    public void run()
17    {
18       for ( int i = valorInicial; i < valorInicial + 3; i++ )
19       {
20          arregloSimpleCompartido.agregar( i ); // agrega un elemento al arreglo compartido
21       } // fin de for
22    } // fin del método run
23 } // fin de la clase EscritorArreglo
```

Figura 23.8 | Agrega enteros a un arreglo compartido con otros objetos `Runnable`.

La clase **PruebaArregloCompartido**

La clase `PruebaArregloCompartido` ejecuta dos tareas `EscritorArreglo` que agregan valores a un solo objeto `ArregloSimple`. En la línea 12 se construye un objeto `ArregloSimple` con seis elementos. En las líneas 15 y 16 se crean dos nuevas tareas `EscritorArreglo`, una que coloca los valores 1 a 3 en el objeto `ArregloSimple`, y una que coloca los valores 11 a 13. En las líneas 19 a 21 se crea un objeto `ExecutorService` y se ejecutan los dos objetos `EscritorArreglo`. En la línea 23 se invoca el método `shutDown` de `ExecutorService` para evitar que se inicien tareas adicionales, y para permitir que la aplicación termine cuando las tareas actuales en ejecución se completen.

```
1  // Fig 23.9: PruebaArregloCompartido.java
2  // Ejecuta dos objetos Runnable para agregar elementos a un objeto ArregloSimple
   compartido.
3  import java.util.concurrent.Executors;
4  import java.util.concurrent.ExecutorService;
5  import java.util.concurrent.TimeUnit;
6
7  public class PruebaArregloCompartido
8  {
9     public static void main( String[] arg )
10    {
```

Figura 23.9 | Ejecuta dos objetos `Runnable` para insertar valores en un arreglo compartido. (Parte 1 de 2).

```
11         // construye el objeto compartido
12         ArregloSimple arregloSimpleCompartido = new ArregloSimple( 6 );
13 
14         // crea dos tareas para escribir en el objeto ArregloSimple compartido
15         EscritorArreglo escritor1 = new EscritorArreglo( 1, arregloSimpleCompartido );
16         EscritorArreglo escritor2 = new EscritorArreglo( 11, arregloSimpleCompartido );
17 
18         // ejecuta las tareas con un objeto ExecutorService
19         ExecutorService ejecutor = Executors.newCachedThreadPool();
20         ejecutor.execute( escritor1 );
21         ejecutor.execute( escritor2 );
22 
23         ejecutor.shutdown();
24 
25         try
26         {
27            // espera 1 minuto para que ambos escritores terminen de ejecutarse
28            boolean tareasTerminaron = ejecutor.awaitTermination(
29               1, TimeUnit.MINUTES );
30 
31            if ( tareasTerminaron )
32               System.out.println( arregloSimpleCompartido ); // imprime el contendio
33            else
34               System.out.println(
35                  "Se agoto el tiempo esperando a que las tareas terminaran." );
36         } // fin de try
37         catch ( InterruptedException ex )
38         {
39            System.out.println(
40               "Hubo una interrupcion mientras esperaba a que terminaran las tareas." );
41         } // fin de catch
42      } // fin de main
43   } // fin de la clase PruebaArregloCompartido
```

```
pool-1-thread-1 escribio  1 en el elemento 0.
Siguiente indice de escritura: 1
pool-1-thread-1 escribio  2 en el elemento 1.
Siguiente indice de escritura: 2
pool-1-thread-1 escribio  3 en el elemento 2.
Siguiente indice de escritura: 3
pool-1-thread-2 escribio 11 en el elemento 0.
Siguiente indice de escritura: 4
pool-1-thread-2 escribio 12 en el elemento 4.
Siguiente indice de escritura: 5
pool-1-thread-2 escribio 13 en el elemento 5.
Siguiente indice de escritura: 6

Contenido de ArregloSimple:
11 2 3 0 12 13
```

Primero `pool-1-thread-1` escribió el valor `1` en el elemento `0`. Después `pool-1-thread-2` escribió el valor `11` en el elemento `0`, con lo cual sobrescribió el valor previamente almacenado.

Figura 23.9 | Ejecuta dos objetos `Runnable` para insertar valores en un arreglo compartido. (Parte 2 de 2).

Recuerde que el método `shutdown` de `ExecutorService` regresa de inmediato. Por ende, cualquier código que aparezca después de la llamada al método `shutdown` de `ExecutorService` en la línea 23 seguirá ejecutándose, siempre y cuando el subproceso `main` siga asignado a un procesador. Nos gustaría imprimir el objeto `ArregloSimple` para mostrarle los resultados *después* de que los subprocesos completan sus tareas. Entonces, necesitamos que el programa espere a que los subprocesos se completen antes de que `main` imprima el contenido del objeto `ArregloSimple`. La interfaz `ExecutorService` proporciona el método **`awaitTermination`** para este fin. Este método devuelve el control al que lo llamó, ya sea cuando se completen todas las tareas que se ejecutan

en el objeto `ExecutorService`, o cuando se agote el tiempo de inactividad especificado. Si todas las tareas se completan antes de que se agote el tiempo de `awaitTermination`, este método devuelve `true`; en caso contrario devuelve `false`. Los dos argumentos para `awaitTermination` representan un valor de límite de tiempo y una unidad de medida especificada con una constante de la clase `TimeUnit` (en este caso, `TimeUnit.MINUTES`). En este ejemplo, si ambas tareas se completan antes de que se agote el tiempo de `awaitTermination`, en la línea 32 se muestra el contenido del objeto `ArregloSimple`. En caso contrario, en las líneas 34 y 35 se imprime un mensaje indicando que las tareas no terminaron de ejecutarse antes de que se agotara el tiempo de `awaitTermination`.

Los resultados en la figura 23.9 demuestran los problemas (resaltados en la salida) que se pueden producir debido a la falla en la sincronización del acceso a los datos compartidos. El valor `1` se escribió para el elemento `0`, y más adelante lo sobrescribió el valor `11`. Además, cuando `indiceEscritura` se incrementó a 3, no se escribió nada en ese elemento, como lo indica el `0` en ese elemento del arreglo impreso.

Recuerde que hemos agregado llamadas al método `sleep` de `Thread` entre las operaciones con los datos compartidos para enfatizar la imprevisibilidad de la programación de subprocesos, y para incrementar la probabilidad de producir una salida errónea. Es importante observar que, aun si estas operaciones pudieran proceder a su ritmo normal, de todas formas veríamos errores en la salida del programa. Sin embargo, los procesadores modernos pueden manejar las operaciones simples del método agregar de `ArregloSimple` con tanta rapidez que tal vez no veríamos los errores ocasionados por los dos subprocesos que ejecutan este método en forma concurrente, aun si probáramos el programa docenas de veces. Uno de los retos de la programación con subprocesamiento múltiple es detectar los errores; pueden ocurrir con tan poca frecuencia que un programa erróneo no producirá resultados incorrectos durante la prueba, creando la ilusión de que el programa es correcto.

23.5.2 Cómo compartir datos con sincronización: hacer las operaciones atómicas

Los errores de la salida de la figura 23.9 pueden atribuirse al hecho de que el objeto compartido (`ArregloSimple`) no es **seguro para los subprocesos**; `ArregloSimple` es susceptible a errores si varios subprocesos lo utilizan en forma concurrente. El problema recae en el método `agregar`, el cual almacena el valor de `indiceEscritura`, coloca un nuevo valor en ese elemento y después incrementa a `indiceEscritura`. Dicho método no presentaría problemas en un programa con un solo subproceso. No obstante, si un subproceso obtiene el valor de `indiceEscritura`, no hay garantía de que otro subproceso no pueda llegar e incrementar `indiceEscritura` antes de que el primer subproceso haya tenido la oportunidad de colocar un valor en el arreglo. Si esto ocurre, el primer subproceso estará escribiendo en el arreglo, con base en un valor pasado de `indiceEscritua`; un valor que ya no sea válido. Otra posibilidad es que un subproceso podría **obtener el valor** de `indiceEscritura` después de que otro subproceso agregue un elemento al arreglo, pero antes de que se incremente `indiceEscritura`. En este caso también, el primer subproceso escribiría en el arreglo con base en un valor inválido para `indiceEscritura`.

`ArregloSimple` no es seguro para los subprocesos, ya que permite que cualquier número de subprocesos lean y modifiquen los datos en forma concurrente, lo cual puede producir errores. Para que `ArregloSimple` sea seguro para los subprocesos, debemos asegurar que no haya dos subprocesos que puedan acceder a este objeto al mismo tiempo. También debemos asegurar que mientras un subproceso se encuentre almacenando `indiceEscritura`, agregando un valor al arreglo e incrementando `indiceEscritura`, ningún otro subproceso pueda leer o cambiar el valor de `indiceEscritura`, o modificar el contenido del arreglo en ningún punto durante estas tres operaciones. En otras palabras, deseamos que estas tres operaciones (almacenar `indiceEscritura`, escribir en el arreglo, incrementar `indiceEscritura`) sean una **operación atómica**, la cual no puede dividirse en suboperaciones más pequeñas. Aunque ningún procesador puede llevar a cabo las tres etapas del método agregar en un solo ciclo de reloj para que la operación sea verdaderamente atómica, podemos simular la atomicidad al asegurar que sólo un subproceso lleve a cabo las tres operaciones al mismo tiempo. Cualquier otro subproceso que necesite realizar la operación deberá esperar hasta que el primer subproceso haya terminado la operación `agregar` en su totalidad.

La atomicidad se puede lograr mediante el uso de la palabra clave `synchronized`. Al colocar nuestras tres suboperaciones en una instrucción `synchronized` o en un método `synchronized`, sólo un subproceso a la vez podrá adquirir el bloqueo y realizar las operaciones. Cuando ese subproceso haya completado todas las operaciones en el bloque `synchronized` y libere el bloqueo, otro subproceso podrá adquirir el bloqueo y empezar a ejecutar las operaciones. Esto asegura que un subproceso que ejecuta las operaciones pueda ver los valores actuales de los datos compartidos, y que estos valores no puedan cambiar de manera inesperada, en medio de las operaciones y como resultado de que otro subproceso los modifique.



Observación de ingeniería de software 23.1

Coloque todos los accesos para los datos mutables que puedan compartir varios subprocesos dentro de instrucciones `synchronized`*, o dentro de métodos* `synchronized` *que puedan sincronizarse con el mismo bloqueo. Al realizar varias operaciones con datos compartidos, mantenga el bloqueo durante toda la operación para asegurar que ésta sea, en efecto, atómica.*

En la figura 23.10 se muestra la clase `ArregloSimple` con la sincronización apropiada. Observe que es idéntica a la clase `ArregloSimple` de la figura 23.7, excepto que agregar es ahora un método `synchronized` (línea 19). Por lo tanto, sólo un subproceso a la vez puede ejecutar este método. Reutilizaremos las clases `EscritorArreglo` (figura 23.8) y `PruebaArregloCompartido` (figura 23.9) del ejemplo anterior.

```
1  // Fig. 23.10: ArregloSimple.java
2  // Clase que administra un arreglo simple para compartirlo entre varios subprocesos.
3  import java.util.Random;
4
5  public class ArregloSimple
6  {
7     private final int arreglo[]; // el arreglo entero compartido
8     private int indiceEscritura = 0; // índice del siguiente elemento a escribir
9     private final static Random generador = new Random();
10
11    // construye un objeto ArregloSimple de un tamaño dado
12    public ArregloSimple( int tamanio )
13    {
14       arreglo = new int[ tamanio ];
15    } // fin del constructor
16
17    // agrega un valor al arreglo compartido
18    public synchronized void agregar( int valor )
19    {
20       int posicion = indiceEscritura; // almacena el índice de escritura
21
22       try
23       {
24          // pone el subproceso en inactividad de 0 a 499 milisegundos
25          Thread.sleep( generador.nextInt( 500 ) );
26       } // fin de try
27       catch ( InterruptedException ex )
28       {
29          ex.printStackTrace();
30       } // fin de catch
31
32       // coloca el valor en el elemento apropiado
33       arreglo[ posicion ] = valor;
34       System.out.printf( "%s escribio %2d en el elemento %d.\n",
35          Thread.currentThread().getName(), valor, posicion );
36
37       ++indiceEscritura; // incrementa el índice del siguiente elemento a escribir
38       System.out.printf( "Siguiente indice de escritura: %d\n", indiceEscritura );
39    } // fin del método agregar
40
41    // se utiliza para imprimir el contenido del arreglo entero compartido
42    public String toString()
43    {
44       String cadenaArreglo = "\nContenido de ArregloSimple:\n";
```

Figura 23.10 | Clase que administra un arreglo simple para compartirlo entre varios subprocesos con sincronización. (Parte 1 de 2).

```
45
46         for ( int i = 0; i < arreglo.length; i++ )
47            cadenaArreglo += arreglo[ i ] + " ";
48
49         return cadenaArreglo;
50      } // fin del método toString
51   } // fin de la clase ArregloSimple
```

```
pool-1-thread-1 escribio  1 en el elemento 0.
Siguiente indice de escritura: 1
pool-1-thread-2 escribio 11 en el elemento 1.
Siguiente indice de escritura: 2
pool-1-thread-2 escribio 12 en el elemento 2.
Siguiente indice de escritura: 3
pool-1-thread-2 escribio  13 en el elemento 3.
Siguiente indice de escritura: 4
pool-1-thread-1 escribio  2 en el elemento 4.
Siguiente indice de escritura: 5
pool-1-thread-1 escribio 3 en el elemento 5.
Siguiente indice de escritura: 6

Contenido de ArregloSimple:
1 11 12 13 2 3
```

Figura 23.10 | Clase que administra un arreglo simple para compartirlo entre varios subprocesos con sincronización. (Parte 2 de 2).

En la línea 18 se declara el método como `synchronized`, lo cual hace que todas las operaciones en este método se comporten como una sola operación atómica. En la línea 20 se realiza la primera suboperación: almacenar el valor de `indiceEscritura`. En la línea 33 se define la segunda suboperación, escribir un elemento en el elemento que está en el índice `posicion`. En la línea 37 se incrementa `indiceEscritura`. Cuando el método termina de ejecutarse en la línea 39, el subproceso en ejecución libera el bloqueo de `ArregloSimple`, lo cual hace posible que otro subproceso empiece a ejecutar el método agregar.

En el método `add sincronizado` mediante la palabra clave `synchronized`, imprimimos mensajes en la consola, indicando el progreso de los subprocesos a medida que ejecutan este método, además de realizar las operaciones actuales requeridas para insertar un valor en el arreglo. Hacemos esto de manera que los mensajes se impriman en el orden correcto, con lo cual usted podrá ver si el método está sincronizado apropiadamente, al comparar estos resultados con los del ejemplo anterior, sin sincronización. En ejemplos posteriores seguiremos imprimiendo mensajes de bloques `synchronized` para fines demostrativos; sin embargo, las operaciones de E/S *no deben* realizarse comúnmente en bloques `synchronized`, ya que es importante minimizar la cantidad de tiempo que un objeto permanece "bloqueado".



Tip de rendimiento 23.2

Mantenga la duración de las instrucciones `synchronized` lo más corta posible, mientras mantiene la sincronización necesaria. Esto minimiza el tiempo de espera para los subprocesos bloqueados. Evite realizar operaciones de E/S, cálculos extensos y operaciones que no requieran sincronización, manteniendo un bloqueo.

Otra observación en relación con la seguridad de los subprocesos: hemos dicho que es necesario sincronizar el acceso a todos los datos que puedan compartirse entre varios subprocesos. En realidad, esta sincronización es necesaria sólo para los **datos mutables**, o los datos que puedan cambiar durante su tiempo de vida. Si los datos compartidos no van a cambiar en un programa con subprocesamiento múltiple, entonces no es posible que un subproceso vea valores antiguos o incorrectos, como resultado de que otro subproceso manipule esos datos.

Al compartir datos inmutables entre subprocesos, declare los correspondientes campos de datos como `final` para indicar que los valores de las variables no cambiarán una vez que se inicialicen. Esto evita que los datos compartidos se modifiquen por accidente más adelante en un programa, lo cual podría comprometer la seguridad

de los subprocesos. Al etiquetar las referencias a los objetos como `final` se indica que la referencia no cambiará, pero no se garantiza que el objeto en sí sea inmutable; esto depende por completo de las propiedades del objeto. Sin embargo, aún es buena práctica marcar las referencias que no cambiarán como `final`, ya que esto obliga al constructor del objeto a ser atómico; el objeto estará construido por completo con todos sus campos inicializados, antes de que el programa lo utilice.



Buena práctica de programación 23.1

Siempre declare los campos de datos que no espera modificar como `final`*. Las variables primitivas que se declaran como* `final` *pueden compartirse de manera segura entre los subprocesos. La referencia a un objeto que se declara como* `final` *asegura que el objeto al que refiere estará completamente construido e inicializado antes de que el programa lo utilice; además, esto evita que la referencia apunte a otro objeto.*

23.6 Relación productor/consumidor sin sincronización

En una **relación productor/consumidor**, la porción correspondiente al **productor** de una aplicación genera datos y los almacena en un objeto compartido, y la porción correspondiente al **consumidor** de una aplicación lee esos datos del objeto compartido. La relación productor/consumidor separa la tarea de identificar el trabajo que se va a realizar de las tareas involucradas en realizar ese trabajo. Un ejemplo de una relación productor/consumidor común es la **cola de impresión**. Aunque una impresora podría no estar disponible si queremos imprimir de una aplicación (el productor), aún podemos "completar" la tarea de impresión, ya que los datos se colocan temporalmente en el disco hasta que la impresora esté disponible. De manera similar, cuando la impresora (consumidor) está disponible, no tiene que esperar hasta que un usuario desee imprimir. Los trabajos en la cola de impresión pueden imprimirse tan pronto como la impresora esté disponible. Otro ejemplo de la relación productor/consumidor es una aplicación que copia datos en CDs, colocándolos en un búfer de tamaño fijo, el cual se vacía a medida que la unidad de CD-RW "quema" los datos en el CD.

En una relación productor/consumidor con subprocesamiento múltiple, un **subproceso productor** genera los datos y los coloca en un objeto compartido, llamado **búfer**. Un **subproceso consumidor** lee los datos del búfer. Esta relación requiere sincronización para asegurar que los valores se produzcan y se consuman de manera apropiada. Todas las operaciones con los datos mutables que comparten varios subprocesos (es decir, los datos en el búfer) deben protegerse con un bloqueo para evitar la corrupción, como vimos en la sección 23.5. Las operaciones con los datos del búfer compartidos por un subproceso productor y un subproceso consumidor son también **dependientes del estado**; las operaciones deben proceder sólo si el búfer se encuentra en el estado correcto. Si el búfer se encuentra en un estado en el que no esté completamente lleno, el productor puede producir; si el búfer se encuentra en un estado en el que no esté completamente vacío, el consumidor puede consumir. Todas las operaciones que acceden al búfer deben usar la sincronización para asegurar que los datos se escriban en el búfer, o se lean del búfer, sólo si éste se encuentra en el estado apropiado. Si el productor que trata de colocar los siguientes datos en el búfer determina que éste se encuentra lleno, el subproceso productor debe esperar hasta que haya espacio para escribir un nuevo valor. Si un subproceso consumidor descubre que el búfer está vacío, o que ya se han leído los datos anteriores, también debe esperar a que haya nuevos datos disponibles.

Considere cómo pueden surgir errores lógicos si no sincronizamos el acceso entre varios subprocesos que manipulan datos compartidos. Nuestro siguiente ejemplo (figuras 23.11 a 23.15) implementa una relación productor/consumidor sin la sincronización apropiada. Un subproceso productor escribe los números del 1 al 10 en un búfer compartido: una sola ubicación de memoria compartida entre dos subprocesos (en este ejemplo, una sola variable `int` llamada `bufer` en la línea 6 de la figura 23.14). El subproceso consumidor lee estos datos del búfer compartido y los muestra en pantalla. La salida del programa muestra los valores que el productor escribe (produce) en el búfer compartido, y los valores que el consumidor lee (consume) del búfer compartido.

Cada valor que el subproceso productor escribe en el búfer compartido lo debe consumir exactamente una vez el subproceso consumidor. Sin embargo y por error, los subprocesos en este ejemplo no están sincronizados. Por lo tanto, los datos se pueden perder o desordenar si el productor coloca nuevos datos en el búfer compartido antes de que el consumidor lea los datos anteriores. Además, los datos pueden duplicarse incorrectamente si el consumidor consume datos de nuevo, antes de que el productor produzca el siguiente valor. Para mostrar estas posibilidades, el subproceso consumidor en el siguiente ejemplo mantiene un total de todos los valores que lee. El subproceso productor produce valores del 1 al 10. Si el consumidor lee cada valor producido una, y sólo una vez, el total será de 55. No obstante, si ejecuta este programa varias veces, podrá ver que el total no es siempre 55

(como se muestra en los resultados de la figura 23.10). Para enfatizar este punto, los subprocesos productor y consumidor en el ejemplo quedan inactivos durante intervalos aleatorios de hasta tres segundos, entre la realización de sus tareas. Por ende, no sabemos cuándo el subproceso productor tratará de escribir un nuevo valor, o cuándo el subproceso consumidor tratará de leer uno.

El programa consiste en la interfaz `Bufer` (figura 23.11) y cuatro clases: `Productor` (figura 23.12), `Consumidor` (figura 23.13), `BuferSinSincronizacion` (figura 23.14) y `PruebaBuferCompartido` (figura 23.15). La interfaz `Bufer` declara los métodos `establecer` (línea 6) y `obtener` (línea 9) que un objeto `Bufer` debe implementar para permitir que el subproceso `Productor` coloque un valor en el `Bufer` y el proceso `Consumidor` obtenga un valor del `Bufer`, respectivamente. Algunos programadores prefieren llamar a estos métodos poner (`put`) y tomar (`take`), respectivamente. En posteriores ejemplos, los métodos `establecer` y `obtener` llamarán métodos que lanzan excepciones `InterruptedException`. Aquí declaramos a cada uno de estos métodos con una cláusula throws, para no tener que modificar esta interfaz para los ejemplos posteriores. En la figura 23.14 se muestra la implementación de esta interfaz.

```
 1  // Fig. 23.11: Bufer.java
 2  // La interfaz Bufer especifica los métodos que el Productor y el Consumidor llaman.
 3  public interface Bufer
 4  {
 5     // coloca valor int value en Bufer
 6     public void establecer( int valor ) throws InterruptedException;
 7
 8     // obtiene valor int de Bufer
 9     public int obtener() throws InterruptedException;
10  } // fin de la interfaz Bufer
```

Figura 23.11 | La interfaz Bufer especifica los métodos que el `Productor` y el `Consumidor` llaman.

La clase `Productor` (figura 23.12) implementa a la interfaz `Runnable`, lo cual le permite ejecutarse como una tarea en un subproceso separado. El constructor (líneas 11 a 14) inicializa la referencia `Bufer` llamada `ubicacionCompartida` con un objeto creado en `main` (línea 14 de la figura 23.15), y que se pasa al constructor en el parámetro `compartido`. Como veremos, éste es un objeto `BuferSinSincronizacion` que implementa a la interfaz `Bufer` sin sincronizar el acceso al objeto compartido. El subproceso `Productor` en este programa ejecuta las tareas especificadas en el método `run` (líneas 17 a 39). Cada iteración del ciclo (líneas 21 a 35) invoca al método `sleep` de `Thread` (línea 25) para colocar el subproceso `Productor` en el estado *en espera sincronizado* durante un intervalo de tiempo aleatorio entre 0 y 3 segundos. Cuando el subproceso despierta, en la línea 26 se pasa el valor de la variable de `control` cuenta al método establecer del objeto `Bufer` para `establecer` el valor del búfer compartido. En la línea 27 se mantiene un total de todos los valores producidos hasta ahora, y en la línea 28 se imprime ese valor. Cuando el ciclo termina, en las líneas 37 y 38 se muestra un mensaje indicando que el `Productor` ha dejado de producir datos, y está terminando. A continuación, el método `run` termina, lo cual indica que el `Productor` completó su tarea. Es importante observar que cualquier método que se llama desde el método `run` de `Runnable` (por ejemplo, el método `establecer` de `Bufer`) se ejecuta como parte del subproceso de ejecución de esa tarea. Este hecho se vuelve importante en la sección 23.7, en donde agregamos la sincronización a la relación productor/consumidor.

```
 1  // Fig. 23.12: Productor.java
 2  // Productor con un método run que inserta los valores del 1 al 10 en el búfer.
 3  import java.util.Random;
 4
 5  public class Productor implements Runnable
 6  {
```

Figura 23.12 | `Productor` con un método `run` que inserta los valores del 1 al 10 en el búfer. (Parte 1 de 2).

```
7       private final static Random generador = new Random();
8       private final Bufer ubicacionCompartida; // referencia al objeto compartido
9
10      // constructor
11      public Productor( Bufer compartido )
12      {
13          ubicacionCompartida = compartido;
14      } // fin del constructor de Productor
15
16      // almacena valores del 1 al 10 en ubicacionCompartida
17      public void run()
18      {
19         int suma = 0;
20
21         for ( int cuenta = 1; cuenta <= 10; cuenta++ )
22         {
23            try // permanece inactivo de 0 a 3 segundos, después coloca valor en Bufer
24            {
25               Thread.sleep( generador.nextInt( 3000 ) ); // periodo de inactividad
                 aleatorio
26               ubicacionCompartida.establecer( cuenta ); // establece el valor en el búfer
27               suma += cuenta; // incrementa la suma de los valores
28               System.out.printf( "\t%2d\n", suma );
29            } // fin de try
30            // si las líneas 25 o 26 se interrumpen, imprime el rastreo de la pila
31            catch ( InterruptedException excepcion )
32            {
33               excepcion.printStackTrace();
34            } // fin de catch
35         } // fin de for
36
37         System.out.println(
38            "Productor termino de producir\nTerminando Productor" );
39      } // fin del método run
40   } // fin de la clase Productor
```

Figura 23.12 | `Productor` con un método `run` que inserta los valores del 1 al 10 en el búfer. (Parte 2 de 2).

La clase `Consumidor` (figura 23.13) también implementa a la interfaz `Runnable`, lo cual permite al `Consumidor` ejecutarse en forma concurrente con el `Productor`. El constructor (líneas 11 a 14) inicializa la referencia `Bufer` llamada `ubicacionCompartida` con un objeto que implementa a la interfaz `Bufer` creada en `main` (figura 23.15), y se pasa al constructor como el parámetro `compartido`. Como veremos, éste es el mismo objeto `BuferSinSincronizacion` que se utiliza para inicializar el objeto `Productor`; por ende, los dos subprocesos comparten el mismo objeto. El subproceso `Consumidor` en este programa realiza las tareas especificadas en el método `run` (líneas 17 a 39). El ciclo en las líneas 21 a 35 itera 10 veces. Cada iteración invoca al método `sleep` de `Thread` (línea 26) para poner al subproceso `Consumidor` en el estado *en espera sincronizado* durante un tiempo máximo de hasta 3 segundos. A continuación, en la línea 27 se utiliza el método obtener de `Bufer` para `obtener` el valor en el búfer compartido, y después se `suma` el valor a la variable suma. En la línea 28 se muestra el total de todos los valores consumidos hasta ese momento. Cuando el ciclo termina, en las líneas 37 y 38 se muestra una línea indicando la suma de los valores consumidos. Después el método `run` termina, lo cual indica que el `Consumidor` completó su tarea. Una vez que ambos subprocesos entran al estado *terminado*, el programa termina.

[*Nota:* llamamos al método `sleep` en el método `run` de las clases `Productor` y `Consumidor` para enfatizar el hecho de que en las aplicaciones con subprocesamiento múltiple, es impredecible saber cuándo va a realizar su tarea cada subproceso, y por cuánto tiempo realizará la tarea cuando tenga un procesador. Por lo general, estas cuestiones de programación de subprocesos son responsabilidad del sistema operativo de la computadora, lo cual está más allá del control del desarrollador de Java. En este programa, las tareas de nuestro subproceso son bastante simples: el `Productor` escribe los valores `1` a `10` en el búfer, y el `Consumidor` lee 10 valores del búfer y `suma` cada

```
// Fig. 23.13: Consumidor.java
// Consumidor con un método run que itera y lee 10 valores del búfer.
import java.util.Random;

public class Consumidor implements Runnable
{
   private final static Random generador = new Random();
   private final Bufer ubicacionCompartida; // referencia al objeto compartido

   // constructor
   public Consumidor( Bufer compartido )
   {
      ubicacionCompartida = compartido;
   } // fin del constructor de Consumidor

   // lee el valor de ubicacionCompartida 10 veces y suma los valores
   public void run()
   {
      int suma = 0;

      for ( int cuenta = 1; cuenta <= 10; cuenta++ )
      {
         // permanece inactivo de 0 a 3 segundos, lee un valor del búfer y lo agrega a suma
         try
         {
            Thread.sleep( generador.nextInt( 3000 ) );
            suma += ubicacionCompartida.obtener();
            System.out.printf( "t\t\t%2d\n", suma );
         } // fin de try
         // si las líneas 26 o 27 se interrumpen, imprime el rastreo de la pila
         catch ( InterruptedException excepcion )
         {
            excepcion.printStackTrace();
         } // fin de catch
      } // fin de for

      System.out.printf( "\n%s %d\n%s\n",
         "Consumidor leyo valores, el total es", suma, "Terminando Consumidor" );
   } // fin del método run
} // fin de la clase Consumidor
```

Figura 23.13 | `Consumidor` con un método `run` que itera y lee 10 valores del búfer.

valor a la variable `suma`. Sin la llamada al método `sleep`, y si el `Productor` se ejecuta primero, dado que hoy en día existen procesadores increíblemente rápidos, es muy probable que el `Productor` complete su tarea antes de que el `Consumidor` tenga oportunidad de ejecutarse. Si el `Consumidor` se ejecutara primero, probablemente consumiría datos basura diez veces y después terminaría antes de que el `Productor` pudiera producir el primer valor real.]

La clase `BuferSinSincronizacion` (figura 23.14) implementa a la interfaz `Bufer` (línea 4). Un objeto de esta clase se comparte entre el `Productor` y el `Consumidor`. En la línea 6 se declara la variable de instancia `bufer` y se inicializa con el valor `-1`. Este valor se utiliza para demostrar el caso en el que el `Consumidor` intenta consumir un valor antes de que el `Productor` coloque siquiera un valor en `bufer`. Los métodos `establecer` (líneas 9 a 13) y `obtener` (líneas 16 a 20) no sincronizan el acceso al campo `bufer`. El método `establecer` simplemente asigna su argumento a `bufer` (línea 12), y el método `obtener` simplemente devuelve el valor de `bufer` (línea 19).

La clase `PruebaBuferCompartido` contiene el método `main` (líneas 9 a 25). En la línea 11 se crea un objeto `ExecutorService` para ejecutar los objetos `Runnable Productor` y `Consumidor`. En la línea 14 se crea un objeto `BuferSinSincronizacion` y se asigna a la variable `Bufer` llamada `ubicacionCompartida`. Este objeto

```
// Fig. 23.14: BuferSinSincronizacion.java
// BuferSinSincronizacion mantiene el entero compartido que utilizan los
// subprocesos productor y consumidor mediante los métodos establecer y obtener.
public class BuferSinSincronizacion implements Bufer
{
   private int bufer = -1; // compartido por los subprocesos productor y consumidor

   // coloca el valor en el búfer
   public void establecer( int valor ) throws InterruptedException
   {
      System.out.printf( "Productor escribe\t%2d", valor );
      bufer = valor;
   } // fin del método establecer

   // devuelve el valor del búfer
   public int obtener() throws InterruptedException
   {
      System.out.printf( "Consumidor lee\t\t%2d", bufer );
      return bufer;
   } // fin del método obtener
} // fin de la clase BuferSinSincronizacion
```

Figura 23.14 | `BuferSinSincronizacion` mantiene el entero compartido que utilizan los subprocesos productor y consumidor mediante los métodos `establecer` y `obtener`.

almacena los datos que compartirán los subprocesos `Productor` y `Consumidor`. En las líneas 23 y 24 se crean y ejecutan los objetos `Productor` y `Consumidor`. Observe que los constructores de `Productor` y `Consumidor` reciben el mismo objeto `Bufer` (`ubicacionCompartida`), por lo que cada objeto se inicializa con una referencia al mismo objeto `Bufer`. Estas líneas también inician de manera implícita los subprocesos, y llaman al método `run` de cada objeto `Runnable`. Por último, en la línea 26 se hace una llamada al método `shutdown`, de manera que la aplicación pueda terminar cuando los subprocesos que ejecutan al `Productor` y al `Consumidor` completen sus tareas. Cuando `main` termina (línea 27), el subproceso principal de ejecución entra al estado *terminado*. De acuerdo con las generalidades de este ejemplo, nos gustaría que el `Productor` se ejecutara primero, y que el `Consumidor` consumiera cada valor producido por el `Productor` sólo una vez. Sin embargo, al estudiar los primeros resultados de la figura 23.15, podemos ver que el `Productor` escribe los valores `1`, `2` y `3` antes de que el `Consumidor` lea su primer valor (`3`). Por lo tanto, los valores `1` y `2` se pierden. Más adelante se pierden los valores `5`, `6` y `9`, mientras que `7` y `8` se leen dos veces y `10` se lee cuatro veces. Así, los primeros resultados producen un total incorrecto de 77, en vez del total correcto de 55. En el segundo conjunto de resultados, el `Consumidor` lee el valor `-1` antes de que el `Productor` escriba siquiera un valor. El `Consumidor` lee el valor `1` cinco veces antes de que el `Productor` escriba el valor `2`. Mientras tanto, los valores `5`, `7`, `8`, `9` y `10` se pierden todos; los últimos cuatro debido a que el `Consumidor` termina antes que el `Productor`. Se muestra un total incorrecto del consumidor de 19. (Las líneas en la salida, en donde el `Productor` o el `Consumidor` han actuado fuera de orden, aparecen resaltadas). Este ejemplo demuestra con claridad que el acceso a un objeto compartido por parte de subprocesos concurrentes se debe controlar con cuidado, ya que de no ser así, un programa podría producir resultados incorrectos.

Para resolver los problemas de los datos perdidos y duplicados, en la sección 23.7 se presenta un ejemplo en el que usamos un objeto `ArrayBlockingQueue` (del paquete `java.util.concurrent`) para sincronizar el acceso al objeto compartido, con lo cual se garantiza que cada uno de los valores se procesará una, y sólo una vez.

```
1  // Fig. 23.15: PruebaBuferCompartido.java
2  // Aplicación con dos subprocesos que manipulan un búfer sin sincronización.
3  import java.util.concurrent.ExecutorService;
4  import java.util.concurrent.Executors;
```

Figura 23.15 | Aplicación con dos subprocesos que manipulan un búfer sin sincronización. (Parte 1 de 3).

```java

public class PruebaBuferCompartido
{
   public static void main( String[] args )
   {
      // crea nueva reserva de subprocesos con dos subprocesos
      ExecutorService aplicacion = Executors.newCachedThreadPool();

      // crea objeto BuferSinSincronizacion para almacenar valores int
      Bufer ubicacionCompartida = new BuferSinSincronizacion();

      System.out.println(
         "Accion\t\t\tValor\tSuma producidos\tSuma consumidos" );
      System.out.println(
         "------\t\t\t-----\t---------------\t---------------\n" );

      // ejecuta el Productor y el Consumidor; a cada uno de ellos
      // le proporciona acceso a ubicacionCompartida
      aplicacion.execute( new Productor( ubicacionCompartida ) );
      aplicacion.execute( new Consumidor( ubicacionCompartida ) );

      aplicacion.shutdown(); // termina la aplicación cuando se completan las tareas
   } // fin de main
} // fin de la clase PruebaBuferCompartido
```

```
Accion                  Valor   Suma producidos   Suma consumidos
------                  -----   ---------------   ---------------

Productor escribe         1       1
Productor escribe         2       3                               ___ 1  se pierde
Productor escribe         3       6                               ___ 2  se pierde
Consumidor lee            3                             3
Productor escribe         4      10
Consumidor lee            4                             7
Productor escribe         5      15
Productor escribe         6      21                               ___ 5  se pierde
Productor escribe         7      28                               ___ 5  se pierde
Consumidor lee            7                            14
Consumidor lee            7                            21         ___ 7  se lee de nuevo
Productor escribe         8      36
Consumidor lee            8                            29
Consumidor lee            8                            37         ___ 8  se lee de nuevo
Productor escribe         9      45
Productor escribe        10      55                               ___ 9  se pierde

Productor termino de producir
Terminando Productor
Consumidor lee           10                            47
Consumidor lee           10                            57         ___ 10  se lee de nuevo
Consumidor lee           10                            67         ___ 10  se lee de nuevo
Consumidor lee           10                            77         ___ 10  se lee de nuevo

Consumidor leyo valores, el total es 77
Terminando Consumidor
```

Figura 23.15 | Aplicación con dos subprocesos que manipulan un búfer sin sincronización. (Parte 2 de 3).

```
Accion                  Valor   Suma producidos   Suma consumidos
------                  -----   ---------------   ---------------
Consumidor lee           -1                              -1   lee -1 datos incorrectos
Productor escribe         1       1
Consumidor lee            1                               0
Consumidor lee            1                               1   1 se lee de nuevo
Consumidor lee            1                               2   1 se lee de nuevo
Consumidor lee            1                               3   1 se lee de nuevo
Consumidor lee            1                               4   1 se lee de nuevo
Productor escribe         2       3
Consumidor lee            2                               6
Productor escribe         3       6
Consumidor lee            3                               9
Productor escribe         4      10
Consumidor lee            4                              13
Productor escribe         5      15
Productor escribe         6      21                           5 se pierde
Consumidor lee            6                              19

Consumidor leyo valores, el total es 19
Terminando Consumidor
Productor escribe         7      28                           7 nunca se lee
Productor escribe         8      36                           8 nunca se lee
Productor escribe         9      45                           9 nunca se lee
Productor escribe        10      55                           10 nunca se lee

Productor termino de producir
Terminando Productor
```

Figura 23.15 | Aplicación con dos subprocesos que manipulan un búfer sin sincronización. (Parte 3 de 3).

23.7 Relación productor/consumidor: ArrayBlockingQueue

Una manera de sincronizar los subprocesos productor y consumidor es utilizar las clases del paquete de concurrencia de Java, el cual encapsula la sincronización por nosotros. Java incluye la clase **ArrayBlockingQueue** (del paquete java.util.concurrent); una clase de búfer completamente implementada, segura para los subprocesos, que implementa a la interfaz **BlockingQueue**. Esta interfaz extiende a la interfaz Queue que vimos en el capítulo 19 y declara los métodos **put** y **take**, los equivalentes con bloqueo de los métodos offer y poll de Queue, respectivamente. El método put coloca un elemento al final del objeto BlockingQueue, y espera si la cola está llena. El método take elimina un elemento de la parte inicial del objeto BlockingQueue, y espera si la cola está vacía. Estos métodos hacen que la clase ArrayBlockingQueue sea una buena opción para implementar un búfer compartido. Debido a que el método put bloquea hasta que haya espacio en el búfer para escribir datos, y el método take bloquea hasta que haya nuevos datos para leer, el productor debe producir primero un valor, el consumidor sólo consume correctamente hasta después de que el productor escribe un valor, y el productor produce correctamente el siguiente valor (después del primero) sólo hasta que el consumidor lea el valor anterior (o primero). ArrayBlockingQueue almacena los datos compartidos en un arreglo. El tamaño de este arreglo se especifica como argumento para el constructor de ArrayBlockingQueue. Una vez creado, un objeto ArrayBlockingQueue tiene su tamaño fijo y no se expandirá para dar cabida a más elementos.

El programa de las figuras 23.16 y 23.17 demuestra a un Productor y un Consumidor accediendo a un objeto ArrayBlockingQueue. La clase BuferBloqueo (figura 23.16) utiliza un objeto ArrayBlockingQueue que almacena un objeto Integer (línea 7). En la línea 11 se crea el objeto ArrayBlockingQueue y se pasa 1 al constructor, para que el objeto contenga un solo valor, como hicimos con el objeto BuferSinSincronizacion de la figura 23.14. Observe que en las líneas 7 y 11 utilizamos genéricos, los cuales se describen en los capítulos 18 y 19. En la sección 23.9 hablaremos sobre los búferes con varios elementos. Debido a que nuestra clase BuferBloqueo utiliza la clase ArrayBlockingQueue (segura para los subprocesos) para administrar el objeto al búfer compartido, BuferBloqueo es en sí segura para los subprocesos, aun cuando no hemos implementado la sincronización nosotros mismos.

```
1  // Fig. 23.16: BuferBloqueo.java
2  // Crea un búfer sincronizado, usando la clase ArrayBlockingQueue.
3  import java.util.concurrent.ArrayBlockingQueue;
4
5  public class BuferBloqueo implements Bufer
6  {
7     private final ArrayBlockingQueue<Integer> bufer; // bufer compartido
8
9     public BuferBloqueo()
10    {
11       bufer = new ArrayBlockingQueue<Integer>( 1 );
12    } // fin del constructor de BuferBloqueo
13
14    // coloca un valor en el búfer
15    public void establecer( int valor ) throws InterruptedException
16    {
17       bufer.put( valor ); // coloca el valor en el búfer
18       System.out.printf( "%s%2d\t%s%d\n", "Productor escribe ", valor,
19          "Celdas de Bufer ocupadas: ", bufer.size() );
20    } // fin del método establecer
21
22    // devuelve el valor del búfer
23    public int obtener() throws InterruptedException
24    {
25       int valorLeido = 0; // inicializa el valor leído del búfer
26
27       valorLeido = bufer.take(); // elimina el valor del búfer
28       System.out.printf( "%s  %2d\t%s%d\n", "Consumidor lee ",
29          valorLeido, "Celdas de Bufer ocupadas: ", bufer.size() );
30
31       return valorLeido;
32    } // fin del método obtener
33 } // fin de la clase BuferBloqueo
```

Figura 23.16 | Crea un búfer sincronizado, usando la clase `ArrayBlockingQueue`.

`BuferBloqueo` implementa a la interfaz `Bufer` (figura 23.11) y utiliza las clases `Productor` (figura 23.12 modificada para eliminar la línea 28) y `Consumidor` (figura 23.13 modificada para eliminar la línea 28) del ejemplo en la sección 23.6. Este método demuestra que los subprocesos que acceden al objeto compartido no están conscientes de que los accesos a su búfer están ahora sincronizados. La sincronización se maneja por completo en los métodos `establecer` y `obtener` de `BuferBloqueo`, al llamar a los métodos sincronizados `put` y `take` de `ArrayBlockingQueue`, respectivamente. Así, los objetos `Runnable Productor` y `Consumidor` están sincronizados en forma apropiada, con sólo llamar a los métodos `establecer` y `obtener` del objeto compartido.

En la línea 17, en el método `establecer` (líneas 15 a 20) se hace una llamada al método `put` del objeto `ArrayBlockingQueue`. La llamada a este método realiza un bloqueo (si es necesario) hasta que haya espacio en el `bufer` para colocar el `valor`. El método `obtener` (líneas 23 a 32) llama al método `take` del objeto `ArrayBlockingQueue` (línea 27). La llamada a este método realiza un bloqueo (si es necesario) hasta que haya un elemento en el `bufer` que se pueda eliminar. En las líneas 18 y 19, y en las líneas 28 y 29, se utiliza el método `size` del objeto `ArrayBlockingQueue` para mostrar el número total de elementos que se encuentran actualmente en el objeto `ArrayBlockingQueue`.

La clase `PruebaBuferBloqueo` (figura 23.17) contiene el método `main` que inicia la aplicación. En la línea 11 se crea un objeto `ExecutorService`, y en la línea 14 se crea un objeto `BuferBloqueo` y se asigna su referencia a la variable `Bufer` llamada `ubicacionCompartida`. En las líneas 16 y 17 se ejecutan los objetos `Runnable Productor` y `Consumidor`. En la línea 19 se hace una llamada al método `shutdown` para terminar la aplicación cuando los subprocesos terminen de ejecutar las tareas `Productor` y `Consumidor`.

Aunque los métodos `put` y `take` de `ArrayBlockingQueue` están sincronizados en forma apropiada, los métodos `establecer` y `obtener` de `BuferBloqueo` (figura 23.16) no se declaran como sincronizados. Por ende,

```java
// Fig 23.17: PruebaBuferBloqueo.java
// Muestra a dos subprocesos manipulando un búfer con bloqueo.
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class PruebaBuferBloqueo
{
   public static void main( String[] args )
   {
      // crea nueva reserva de subprocesos con dos subprocesos
      ExecutorService aplicacion = Executors.newCachedThreadPool();

      // crea objeto BuferBloqueo para almacenar valores int
      Bufer ubicacionCompartida = new BuferBloqueo();

      aplicacion.execute( new Productor( ubicacionCompartida ) );
      aplicacion.execute( new Consumidor( ubicacionCompartida ) );

      aplicacion.shutdown();
   } // fin de main
} // fin de la clase PruebaBuferBloqueo
```

```
Productor escribe   1       Celdas de Bufer ocupadas: 1
Consumidor lee      1       Celdas de Bufer ocupadas: 0
Productor escribe   2       Celdas de Bufer ocupadas: 1
Consumidor lee      2       Celdas de Bufer ocupadas: 0

Productor escribe   3       Celdas de Bufer ocupadas: 1
Consumidor lee      3       Celdas de Bufer ocupadas: 0
Productor escribe   4       Celdas de Bufer ocupadas: 1
Consumidor lee      4       Celdas de Bufer ocupadas: 0
Productor escribe   5       Celdas de Bufer ocupadas: 1
Consumidor lee      5       Celdas de Bufer ocupadas: 0
Productor escribe   6       Celdas de Bufer ocupadas: 1
Consumidor lee      6       Celdas de Bufer ocupadas: 0
Productor escribe   7       Celdas de Bufer ocupadas: 1
Consumidor lee      7       Celdas de Bufer ocupadas: 0
Productor escribe   8       Celdas de Bufer ocupadas: 1
Consumidor lee      8       Celdas de Bufer ocupadas: 0
Productor escribe   9       Celdas de Bufer ocupadas: 1
Consumidor lee      9       Celdas de Bufer ocupadas: 0
Productor escribe  10       Celdas de Bufer ocupadas: 1

Productor termino de producir
Terminando Productor
Consumidor lee     10       Celdas de Bufer ocupadas: 0

Consumidor leyo valores, el total es 55
Terminando Consumidor
```

Figura 23.17 | Muestra a dos subprocesos manipulando un búfer con bloqueo.

las instrucciones que se realizan en el método `establecer` (la operación put en la línea 19, y la salida en las líneas 20 y 21) no son atómicas; tampoco lo son las instrucciones en el método `obtener` (la operación `take` en la línea 36, y la salida en las líneas 37 y 38). Por lo tanto, no hay garantía de que cada operación de salida ocurrirá justo después de la correspondiente operación `put` o `take`, y los resultados pueden aparecer fuera de orden. Aun si lo hacen, el objeto `ArrayBlockingQueue` está sincronizando de manera correcta el acceso a los datos, como se puede ver mediante el hecho de que la suma de los valores que lee el consumidor siempre es correcta.

23.8 Relación productor/consumidor con sincronización

El ejemplo anterior mostró cómo varios subprocesos pueden compartir un búfer de un solo elemento en forma segura para los subprocesos, al utilizar la clase `ArrayBlockingQueue` que encapsula la sincronización necesaria para proteger los datos compartidos. Para fines educativos, ahora le explicaremos cómo puede implementar usted mismo un búfer compartido, usando la palabra clave `synchronized`. El uso de un objeto `ArrayBlockingQueue` producirá código con mejor capacidad de mantenimiento y más rendimiento.

El primer paso para sincronizar el acceso al búfer es implementar los métodos `establecer` y `obtener` como métodos `synchronized`. Esto requiere que un subproceso obtenga el bloqueo de monitor en el objeto `Bufer` antes de tratar de acceder a los datos del búfer, pero no resuelve el problema de dependencia asociado con las relaciones productor/consumidor. Debemos asegurar que los subprocesos procedan con una operación sólo si el búfer se encuentra en el estado apropiado. Necesitamos una manera de permitir a nuestros subprocesos esperar, dependiendo de la validez de ciertas condiciones. En el caso de colocar un nuevo elemento en el búfer, la condición que permite que la operación continúe es que el búfer no esté lleno. En el caso de obtener un elemento del búfer, la condición que permite que la operación continúe es que el búfer no esté vacío. Si la condición en cuestión es verdadera, la operación puede continuar; si es falsa, el subproceso debe esperar hasta que la condición se vuelva verdadera. Cuando un subproceso espera en base a una condición, se elimina de la contención para el procesador, se coloca en la cola de espera del objeto y se libera el bloqueo que contiene.

Los métodos `wait`*,* `notify` *y* `notifyAll`

Los métodos `wait`, `notify` y `notifyAll`, que se declaran en la clase `Object` y son heredados por las demás clases, se pueden usar con condiciones para hacer que los subprocesos esperen cuando no pueden realizar sus tareas. Si un subproceso obtiene el bloqueo de monitor en un objeto, y después determina que no puede continuar con su tarea en ese objeto sino hasta que se cumpla cierta condición, el subproceso puede llamar al método **`wait`** de `Object`; esto libera el bloqueo de monitor en el objeto, y el subproceso queda en el estado *en espera* mientras el otro subproceso trata de entrar a la(s) instrucción(es) o método(s) `synchronized` del objeto. Cuando un subproceso que ejecuta una instrucción (o método) `synchronized` completa o cumple con la condición en la que otro subproceso puede estar esperando, puede llamar al método **`notify`** de `Object` para permitir que un subproceso en espera cambie al estado *ejecutable* de nuevo. En este punto, el subproceso que cambió del estado *en espera* al estado *ejecutable* puede tratar de readquirir el bloqueo de monitor en el objeto. Aun si el subproceso puede readquirir el bloqueo de monitor, tal vez no pueda todavía realizar su tarea en este momento; en ese caso, el subproceso volverá a entrar al estado *en espera* y liberará de manera implícita el bloqueo de monitor. Si un subproceso llama a **`notifyAll`**, entonces todos los subprocesos que esperan el bloqueo de monitor serán candidatos para readquirir el bloqueo (es decir, todos cambian al estado *ejecutable*). Recuerde que sólo un subproceso a la vez puede adquirir el bloqueo de monitor en el objeto; los demás subprocesos que traten de adquirir el mismo bloqueo de monitor estarán *bloqueados* hasta que el bloqueo de monitor esté disponible de nuevo (es decir, hasta que ningún otro subproceso se esté ejecutando en una instrucción `synchronized` en ese objeto).



Error común de programación 23.1

Es un error si un subproceso llama a `wait`*,* `notify` *o* `notifyAll` *en un objeto sin haber adquirido un bloqueo para él. Esto produce una excepción* **`IllegalMonitorStateException`**.



Tip para prevenir errores 23.1

Es una buena práctica utilizar a `notifyAll` *para notificar a los subprocesos en espera para que cambien al estado ejecutable. Al hacer esto, evitamos la posibilidad de que el programa se olvide de los subprocesos en espera, que de otra forma quedarían aplazados indefinidamente (inanición).*

La aplicación de las figuras 23.18 y 23.19 demuestra cómo un `Productor` y un `Consumidor` acceden a un búfer compartido con sincronización. En este caso, el `Productor` siempre produce un valor primero, el `Consumidor` consume correctamente sólo hasta después de que el `Productor` produzca un valor, y el `Productor` produce correctamente el siguiente valor sólo hasta después de que el `Consumidor` consuma el valor anterior (o el primero). Reutilizamos la interfaz `Bufer` y las clases `Productor` y `Consumidor` del ejemplo de la sección 23.6. La sincronización se maneja en los métodos `establecer` y `obtener` de la clase `BuferSincronizado` (figura 23.18), la cual implementa a la interfaz `Bufer` (línea 4). Por ende, los métodos `Productor` y `Consumidor` simplemente llaman a los métodos `synchronized` `establecer` y `obtener` del objeto compartido.

```
// Fig. 23.18: BuferSincronizado.java
// Sincronización del acceso a datos compartidos, usando los
// métodos wait y notify de Object.
public class BuferSincronizado implements Bufer
{
   private int bufer = -1; // compartido por los subprocesos productor y consumidor
   private boolean ocupado = false; // indica si el búfer está ocupado o no

   // coloca el valor en el búfer
   public synchronized void establecer( int valor ) throws InterruptedException
   {
      // mientras no haya ubicaciones vacías, coloca el subproceso en espera
      while ( ocupado )
      {
         // imprime información del subproceso e información del búfer, después espera
         System.out.println( "Productor trata de escribir." );
         mostrarEstado( "Bufer lleno. Productor espera." );
         wait();
      } // fin de while

      bufer = valor; // establece el nuevo valor del búfer

      // indica que el productor no puede almacenar otro valor
      // hasta que el consumidor obtenga el valor actual del búfer
      ocupado = true;

      mostrarEstado( "Productor escribe " + bufer );

      notifyAll(); // indica al (los) subproceso(s) en espera que entren al estado runnable
   } // fin del método establecer; libera el bloqueo sobre BuferSincronizado

   // devuelve el valor del búfer
   public synchronized int obtener() throws InterruptedException
   {
      // mientras no haya datos para leer, coloca el subproceso en el estado en espera
      while ( !ocupado )
      {
         // imprime la información del subproceso y la información del búfer, después espera
         System.out.println( "Consumidor trata de leer." );
         mostrarEstado( "Bufer vacio. Consumidor espera." );
         wait();
      } // fin de while

      // indica que el productor puede almacenar otro valor
      // debido a que el consumidor acaba de obtener el valor del búfer
      ocupado = false;

      mostrarEstado( "Consumidor lee " + bufer );

      notifyAll(); // indica al (los) subproceso(s) en espera que entren al estado runnable

      return bufer;
   } // fin del método obtener; libera el bloqueo sobre BuferSincronizado

```

Figura 23.18 | Sincronización del acceso a datos compartidos, usando los métodos `wait` y `notify` de `Object`. (Parte 1 de 2).

```
55      // muestra la operación actual y el estado del búfer
56      public void mostrarEstado( String operacion )
57      {
58         System.out.printf( "40s%d\t\t%b\n\n", operacion, bufer,
59            ocupado );
60      } // fin del método mostrarEstado
61   } // fin de la clase BuferSincronizado
```

Figura 23.18 | Sincronización del acceso a datos compartidos, usando los métodos `wait` y `notify` de `Object`. (Parte 2 de 2).

Campos y métodos de la clase `BuferSincronizado`

La clase `BuferSincronizado` contiene dos campos: `bufer` (línea 6) y `ocupado` (línea 7). Los métodos `establecer` (líneas 10 a 30) y `obtener` (líneas 33 a 53) se declaran como `synchronized`; sólo un subproceso puede llamar a uno de estos métodos a la vez, en un objeto `BuferSincronizado` específico. El campo `ocupado` se utiliza para determinar si es turno del `Productor` o del `Consumidor` para realizar una tarea. Este campo se utiliza en expresiones condicionales en los métodos `establecer` y `obtener`. Si `ocupado` es `false`, el `bufer` está vacío y el `Consumidor` no puede leer el valor de `bufer`, pero el `Productor` puede colocar un valor en este `bufer`. Si `ocupado` es `true`, el `Consumidor` puede leer un valor de `bufer`, pero el `Productor` no puede colocar un valor en este `bufer`.

El método `establecer` *y el subproceso* `Productor`

Cuando el método `run` del subproceso `Productor` invoca al método `establecer sincronizado`, el subproceso intenta de manera implícita adquirir el bloqueo de monitor del objeto `BuferSincronizado`. Si el bloqueo de monitor está disponible, el subproceso `Productor` adquiere el bloqueo de manera implícita. Después, el ciclo en las líneas 13 a 19 primero determina si `ocupado` es `true`. Si es así, `bufer` está lleno, por lo que en la línea 16 se imprime un mensaje indicando que el subproceso `Productor` está tratando de escribir un valor, y en la línea 17 se invoca al método `mostrarEstado` (líneas 56 a 60) para imprimir otro mensaje, indicando que `bufer` está lleno y que el subproceso `Productor` está esperando hasta que haya espacio. En la línea 18 se invoca al método `wait` (heredado de `Object` por `BuferSincronizado`) para colocar el subproceso que llamó al método `establecer` (es decir, el subproceso `Productor`) en el estado *en espera* para el objeto `BuferSincronizado`. La llamada a `wait` hace que el subproceso que llama libere implícitamente el bloqueo sobre el objeto `BuferSincronizado`. Esto es importante, ya que el subproceso no puede actualmente realizar su tarea, y porque se debe permitir a otros subprocesos (en este caso, el `Consumidor`) acceder al objeto para permitir que cambie la condición (`ocupado`). Ahora, otro subproceso puede tratar de adquirir el bloqueo del objeto `BuferSincronizado` e invocar al método `establecer` u `obtener` del objeto.

El subproceso `Productor` permanece en el estado *en espera* hasta que otro subproceso notifica al `Productor` que puede continuar; en este punto el `Productor` regresa al estado *ejecutable* y trata de readquirir en forma implícita el bloqueo sobre el objeto `BuferSincronizado`. Si el bloqueo está disponible, el subproceso `Productor` readquiere el bloqueo, y el método `establecer` continúa ejecutándose con la siguiente instrucción después de la llamada a `wait`. Como `wait` se llama en un ciclo, la condición de continuación del ciclo se evalúa de nuevo para determinar si el subproceso puede proceder. Si no es así, entonces `wait` se invoca de nuevo; en caso contrario, el método `establecer` continúa con la siguiente instrucción después del ciclo.

En la línea 21, en el método `establecer` se asigna el valor al `bufer`. En la línea 25 se establece `ocupado` en `true` para indicar que el `bufer` ahora contiene un valor (es decir, un consumidor puede leer el valor, pero un `Productor` no puede colocar todavía un valor ahí). En la línea 27 se invoca al método `mostrarEstado` para imprimir un mensaje que indique que el `Productor` está escribiendo un nuevo valor en el `bufer`. En la línea 29 se invoca el método `notifyAll` (heredado de `Object`). Si hay subprocesos en espera del bloqueo de monitor del objeto `BuferSincronizado`, esos subprocesos entran al estado *ejecutable* y pueden ahora tratar de readquirir el bloqueo. El método `notifyAll` regresa de inmediato, y el método `establecer` regresa entonces al método que hizo la llamada (es decir, el método `run` de `Productor`). Cuando el método `establecer` regresa, libera de manera implícita el bloqueo de monitor sobre el objeto `BuferSincronizado`.

El método **obtener** *y el subproceso* **Consumidor**

Los métodos `obtener` y `establecer` se implementan de manera similar. Cuando el método `run` del subproceso `Consumidor` invoca al método `obtener sincronizado`, el subproceso trata de adquirir el bloqueo de monitor sobre el objeto `BuferSincronizado`. Si el bloqueo está disponible, el subproceso `Consumidor` lo adquiere. Después, el ciclo while en las líneas 36 a 42 determina si `ocupado` es `false`. Si es así, el búfer está vacío, por lo que en la línea 39 se imprime un mensaje indicando que el subproceso `Consumidor` está tratando de leer un valor, y en la línea 40 se invoca el método `mostrarEstado` para imprimir un mensaje que indique que el búfer está vacío, y que el subproceso `Consumidor` está esperando. En la línea 41 se invoca el método `wait` para colocar el subproceso que llamó al método `obtener` (es decir, el `Consumidor`) en el estado *en espera* del objeto `BuferSincronizado`. De nuevo, la llamada a `wait` hace que el subproceso que hizo la llamada libere de manera implícita el bloqueo sobre el objeto `BuferSincronizado`, para que otro subproceso pueda tratar de adquirir el bloqueo del objeto `BuferSincronizado` e invocar al método establecer u obtener del objeto. Si el bloqueo sobre el objeto `BuferSincronizado` no está disponible (por ejemplo, si el `Productor` no ha regresado todavía del método `establecer`), el `Consumidor` se detiene hasta que el bloqueo esté disponible.

El subproceso `Consumidor` permanece en el estado *en espera* hasta que otro subproceso le notifica que puede continuar; en este punto, el subproceso `Consumidor` regresa al estado *ejecutable* y trata de readquirir en forma implícita el bloqueo sobre el objeto `BuferSincronizado`. Si el bloqueo está disponible, el `Consumidor` readquiere el bloqueo y el método `obtener` continúa ejecutándose con la siguiente instrucción después de `wait`. Debido a que la llamada a `wait` ocurre dentro de un ciclo, la condición de continuación del ciclo se evalúa de nuevo para determinar si el subproceso puede continuar con su ejecución. Si no es así, `wait` se invoca de nuevo; en caso contrario, el método `obtener` continúa con la siguiente instrucción después del ciclo. En la línea 46 se establece la variable `ocupado` en `false`, para indicar que el `bufer` está ahora vacío (es decir, un `Consumidor` no puede leer el valor, pero un `Productor` puede colocar otro valor en el `bufer`), en la línea 48 se hace una llamada al método `mostrarEstado` para indicar que el consumidor está leyendo y en la línea 50 se invoca el método `notifyAll`. Si hay subprocesos en el estado *en espera* del bloqueo sobre este objeto `BuferSincronizado`, entran al estado *ejecutable* y pueden ahora readquirir el bloqueo. El método `notifyAll` regresa de inmediato, y después el método `obtener` devuelve el valor de `bufer` al método que lo llamó. Cuando el método `obtener` regresa, el bloqueo sobre el objeto `BuferSincronizado` se libera de manera implícita.



Tip para prevenir errores 23.2

Siempre debe invocar al método `wait` *en un ciclo que evalúe la condición en base a la cual la tarea está esperando. Es posible que un subproceso vuelva a entrar al estado* ejecutable *(a través de una espera sincronizada o debido a que otro subproceso hace una llamada a* `notifyAll`*) antes de que se cumpla la condición. Al evaluar la condición de nuevo, nos aseguramos que el subproceso no se ejecute por error, si se le notificó antes.*

Prueba de la clase **BuferSincronizado**

La clase `PruebaBuferCompartido2` (figura 23.19) es similar a la clase `PruebaBuferCompartido` (figura 23.15). `PruebaBuferCompartido2` contiene el método `main` (líneas 8 a 24), el cual inicia la aplicación. En la línea 11 se crea un objeto `ExecutorService` para ejecutar las tareas `Productor` y `Consumidor`. En la línea 14 se crea un objeto `BuferSincronizado` y se asigna su referencia a la variable `Bufer` llamada `ubicacionCompartida`. Este objeto almacena los datos que se compartirán entre el `Productor` y el `Consumidor`. En las líneas 16 y 17 se muestran los encabezados de columna para los resultados. En las líneas 20 y 21 se ejecuta un `Productor` y un `Consumidor`. Por último, en la línea 23 se hace una llamada al método `shutdown` para terminar la aplicación cuando el `Productor` y el `Consumidor` completen sus tareas. Cuando el método `main` termina (línea 24), el subproceso principal de ejecución termina.

Estudie los resultados de la figura 23.19. Observe que cada entero producido se consume sólo una vez; no se pierden valores, y no se consumen valores más de una vez. La sincronización asegura que el `Productor` produzca un valor sólo cuando el búfer esté vacío, y que el `Consumidor` consuma sólo cuando el búfer esté lleno. El `Productor` siempre va primero, el `Consumidor` espera si el `Productor` no ha producido desde la última vez que el `Consumidor` consumió, y el `Productor` espera si el `Consumidor` no ha consumido todavía el valor que el `Productor` produjo más recientemente. Ejecute este programa varias veces para confirmar que cada entero producido se consuma sólo una vez. En los resultados de ejemplo, observe las líneas resaltadas que indican cuándo deben esperar el `Productor` y el `Consumidor` para realizar sus respectivas tareas.

```java
// Fig 23.19: PruebaBuferCompartido2.java
// La aplicación muestra cómo dos subprocesos manipulan un búfer sincronizado.
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class PruebaBuferCompartido2
{
   public static void main( String[] args )
   {
      // crea nueva reserva con dos subprocesos
      ExecutorService aplicacion = Executors.newCachedThreadPool();

      // crea objeto BuferSincronizado para almacenar valores int
      Bufer ubicacionCompartida = new BuferSincronizado();

      System.out.printf( "%-40s%s\t\t%s\n%-40s%s\n\n", "Operacion",
         "Bufer", "Ocupado", "---------", "------\t\t--------" );

      // ejecuta las tareas Productor y Consumidor
      aplicacion.execute( new Productor( ubicacionCompartida ) );
      aplicacion.execute( new Consumidor( ubicacionCompartida ) );

      aplicacion.shutdown();
   } // fin de main
} // fin de la clase PruebaBuferCompartido2
```

```
Operacion                               Bufer           Ocupado
---------                               ------          --------

Consumidor trata de leer.
Bufer vacio. Consumidor espera.         -1              false

Productor escribe 1                     1               true

Consumidor lee 1                        1               false

Consumidor trata de leer.
Bufer vacio. Consumidor espera.         1               false

Productor escribe 2                     2               true

Consumidor lee 2                        2               false

Productor escribe 3                     3               true

Consumidor lee 3                        3               false

Productor escribe 4                     4               true

Productor trata de escribir.
Bufer lleno. Productor espera.          4               true

Consumidor lee 4                        4               false

Productor escribe 5                     5               true

Consumidor lee 5                        5               false

Productor escribe 6                     6               true
```

Figura 23.19 | La aplicación muestra cómo dos subprocesos manipulan un búfer sincronizado. (Parte 1 de 2).

```
Productor trata de escribir.
Bufer lleno. Productor espera.             6                     true

Consumidor lee 6                           6                     false

Productor escribe 7                        7                     true

Productor trata de escribir.
Bufer lleno. Productor espera.             7                     true

Consumidor lee 7                           7                     false

Productor escribe 8                        8                     true

Consumidor lee 8                           8                     false

Consumidor trata de leer.
Bufer vacio. Consumidor espera.            8                     false

Productor escribe 9                        9                     true

Consumidor lee 9                           9                     false

Consumidor trata de leer.
Bufer vacio. Consumidor espera.            9                     false

Productor escribe 10                       10                    true

Consumidor lee 10                          10                    false

Productor termino de producir
Terminando Productor

Consumidor leyo valores, el total es 55
Terminando Consumidor
```

Figura 23.19 | La aplicación muestra cómo dos subprocesos manipulan un búfer sincronizado. (Parte 2 de 2).

23.9 Relación productor/consumidor: búferes delimitados

El programa de la sección 23.8 utiliza la sincronización de subprocesos para garantizar que dos subprocesos manipulen correctamente los datos en un búfer compartido. Sin embargo, la aplicación tal vez no tenga un rendimiento óptimo. Si los dos subprocesos operan a distintas velocidades, uno de ellos invertirá más tiempo (o la mayoría de éste) esperando. Por ejemplo, en el programa de la sección 23.8 compartimos una sola variable entera entre los dos subprocesos. Si el subproceso `Productor` produce valores con más rapidez de la que el `Consumidor` puede consumirlos, entonces el subproceso `Productor` espera al `Consumidor`, ya que no hay más ubicaciones en el búfer en donde se pueda colocar el siguiente valor. De manera similar, si el `Consumidor` consume valores con más rapidez de la que el `Productor` los produce, el `Consumidor` espera hasta que el `Productor` coloque el siguiente valor en el búfer compartido. Aun cuando tenemos subprocesos que operan a las mismas velocidades relativas, algunas veces esos subprocesos pueden "salirse de sincronía" durante un periodo de tiempo, lo cual provoca que uno de ellos tenga que esperar al otro. No podemos hacer suposiciones acerca de las velocidades relativas de los subprocesos concurrentes; las interacciones que ocurren con el sistema operativo, la red, el usuario y otros componentes pueden hacer que los subprocesos operen a distintas velocidades. Cuando esto ocurre, los subprocesos esperan. Cuando los subprocesos esperan de manera excesiva, los programas se vuelven menos eficientes, los programas interactivos se vuelven menos responsivos y las aplicaciones sufren retrasos extensos.

Búferes delimitados

Para minimizar la cantidad de tiempo de espera para los subprocesos que comparten recursos y operan a las mismas velocidades promedio, podemos implementar un **bufer delimitado** que proporcione un número fijo de celdas de búfer en las que el `Productor` pueda colocar valores, y de las cuales el `Consumidor` pueda obtener esos valores. (De hecho, ya hemos realizado esto con la clase `ArrayBlockingQueue` en la sección 23.7). Si el `Productor` produce temporalmente valores con más rapidez de la que el `Consumidor` pueda consumirlos, el `Productor` puede escribir otros valores en el espacio adicional del búfer (si hay disponible). Esta capacidad permite al `Productor` realizar su tarea, aún cuando el `Consumidor` no esté listo para obtener el valor que se esté produciendo en ese momento. De manera similar, si el `Consumidor` consume con más rapidez de la que el `Productor` produce nuevos valores, el `Consumidor` puede leer valores adicionales (si los hay) del búfer. Eso permite al `Consumidor` mantenerse ocupado, aún cuando el `Productor` no esté listo para producir valores adicionales.

Observe que, incluso hasta un búfer delimitado es inapropiado si el `Productor` y el `Consumidor` operan consistentemente a distintas velocidades. Si el `Consumidor` siempre se ejecuta con más rapidez que el `Productor`, entonces basta con tener un búfer con una sola ubicación. Las ubicaciones adicionales simplemente desperdiciarían memoria. Si el `Productor` siempre se ejecuta con más rapidez, sólo un búfer con un número "infinito" de ubicaciones podría absorber la producción adicional. No obstante, si el `Productor` y el `Consumidor` se ejecutan aproximadamente a la misma velocidad promedio, un búfer delimitado ayuda a suavizar los efectos de cualquier aceleración o desaceleración ocasional en la ejecución de cualquiera de los dos subprocesos.

La clave para usar un búfer delimitado con un `Productor` y un `Consumidor` que operan aproximadamente a la misma velocidad es proporcionar al búfer suficientes ubicaciones para que pueda manejar la producción "extra" anticipada. Si, durante un periodo de tiempo, determinamos que el `Productor` con frecuencia produce hasta tres valores más de los que el `Consumidor` puede consumir, podemos proporcionar un búfer de por lo menos tres celdas para manejar la producción adicional. Si hacemos el búfer demasiado pequeño, los subprocesos tendrían que esperar más; si hacemos el búfer demasiado grande, se desperdiciaría memoria.



Tip de rendimiento 23.3

Aun cuando se utilice un búfer delimitado, es posible que un subproceso productor pueda llenar el búfer, lo cual obligaría al productor a esperar hasta que un consumidor consumiera un valor para liberar un elemento en el búfer. De manera similar, si el búfer está vacío en algún momento dado, un subproceso consumidor debe esperar hasta que el productor produzca otro valor. La clave para usar un búfer delimitado es optimizar su tamaño para minimizar la cantidad de tiempo de espera de los subprocesos, sin desperdiciar espacio.

Búferes delimitados que utilizan a `ArrayBlockingQueue`

La manera más simple de implementar un búfer delimitado es utilizar un objeto `ArrayBlockingQueue` para el búfer, de manera que se haga cargo de todos los detalles de la sincronización por nosotros. Para ello, podemos reutilizar el ejemplo de la sección 23.7 y pasar simplemente el tamaño deseado para el búfer delimitado al constructor de `ArrayBlockingQueue`. En vez de repetir nuestro ejemplo anterior con `ArrayBlockingQueue` con un tamaño distinto, vamos a presentar un ejemplo que ilustra cómo podemos construir un búfer delimitado por nuestra cuenta. De nuevo, observe que si utilizamos un objeto `ArrayBlockingQueue`, nuestro código tendrá una mejor capacidad de mantenimiento y un mejor rendimiento.

Implementación de su propio búfer delimitado como un búfer circular

El programa de las figuras 23.20 y 23.21 demuestra cómo un `Productor` y un `Consumidor` acceden a un búfer delimitado con sincronización. Implementamos el búfer delimitado en la clase `BuferCircular` (figura 23.20) como un **búfer circular** que utiliza un arreglo compartido de tres elementos. Un búfer circular escribe los elementos de un arreglo y los lee en orden, empezando en la primera celda y avanzando hacia la última. Cuando un `Productor` o `Consumidor` llega al último elemento, regresa al primero y empieza a escribir o leer, respectivamente, de ahí. En esta versión de la relación productor/consumidor, el `Consumidor` consume un valor sólo cuando el arreglo no está vacío y el `Productor` produce un valor sólo cuando el arreglo no está lleno. Las instrucciones que crearon e iniciaron los objetos subproceso en el método `main` de la clase `PruebaBuferCompartido2` (figura 23.19) ahora aparecen en la clase `PruebaBuferCircular` (figura 23.21).

En la línea 5 se inicializa el arreglo `bufer` como un arreglo de tres elementos que representa el búfer circular. La variable `celdasOcupadas` (línea 7) cuenta el número de elementos en `bufer` que contienen datos a leer.

```
// Fig. 23.20: BuferCircular.java
// Sincronización del acceso a un búfer delimitado compartido, con tres elementos.
public class BuferCircular implements Bufer
{
   private final int[] bufer = { -1, -1, -1 }; // búfer compartido

   private int celdasOcupadas = 0; // número de búferes utilizados
   private int indiceEscritura = 0; // índice del siguiente elemento a escribir
   private int indiceLectura = 0; // índice del siguiente elemento a leer

   // coloca un valor en el búfer
   public synchronized void establecer( int valor ) throws InterruptedException
   {
      // imprime información del subproceso y del búfer, después espera;
      // mientras no haya ubicaciones vacías, coloca el subproceso en estado de espera
      while ( celdasOcupadas == bufer.length )
      {
         System.out.printf( "Bufer está lleno. Productor espera.\n" );
         wait(); // espera hasta que haya una celda libre en el búfer
      } // fin de while

      bufer[ indiceEscritura ] = valor; // establece nuevo valor del búfer

      // actualiza índice de escritura circular
      indiceEscritura = ( indiceEscritura + 1 ) % bufer.length;

      ++celdasOcupadas; // una celda más del búfer está llena
      mostrarEstado( "Productor escribe " + valor );
      notifyAll(); // notifica a los subprocesos en espera para que lean del búfer
   } // fin del método establecer

   // devuelve un valor del búfer
   public synchronized int obtener() throws InterruptedException
   {
      // espera hasta que el búfer tenga datos, después lee el valor;
      // mientras no haya datos para leer, coloca el subproceso en estado de espera
      while ( celdasOcupadas == 0 )
      {
         System.out.printf( "Bufer esta vacio. Consumidor espera.\n" );
         wait(); // espera hasta que se llene una celda del búfer
      } // fin de while

      int valorLeido = bufer[ indiceLectura ]; // lee un valor del búfer

      // actualiza índice de lectura circular
      indiceLectura = ( indiceLectura + 1 ) % bufer.length;

      --celdasOcupadas; // hay una celda ocupada menos en el búfer
      mostrarEstado( "Consumidor lee " + valorLeido );
      notifyAll(); // notifica a los subprocesos en espera que pueden escribir en el
      búfer

      return valorLeido;
   } // fin del método obtener

   // muestra la operación actual y estado del búfer
   public void mostrarEstado( String operacion )
   {
      // imprime operación y número de celdas ocupadas del búfer
```

Figura 23.20 | Sincronización del acceso a un búfer delimitado compartido, con tres elementos. (Parte 1 de 2).

```
59         System.out.printf( "%s%s%d)\n%s", operacion,
60            " (celdas ocupadas del bufer: ", celdasOcupadas, "celdas bufer: " );
61
62         for ( int valor : bufer )
63            System.out.printf( " %2d ", valor ); // imprime los valores que hay en el
              búfer
64
65         System.out.print( "\n                 " );
66
67         for ( int i = 0; i < bufer.length; i++ )
68            System.out.print( "---- " );
69
70         System.out.print( "\n                 " );
71
72         for ( int i = 0; i < bufer.length; i++ )
73         {
74            if ( i == indiceEscritura && i == indiceLectura )
75               System.out.print( " WR" ); // indice de escritura y de lectura
76            else if ( i == indiceEscritura )
77               System.out.print( " W   " ); // sólo el índice de escritura
78            else if ( i == indiceLectura )
79               System.out.print( "  R  " ); // sólo el índice de lectura
80            else
81               System.out.print( "     " ); // ningún índice
82         } // fin de for
83
84         System.out.println( "\n" );
85      } // fin del método mostrarEstado
86   } // fin de la clase BuferCircular
```

Figura 23.20 | Sincronización del acceso a un búfer delimitado compartido, con tres elementos. (Parte 2 de 2).

Cuando `buferesOcupados` es `0`, no hay datos en el búfer circular y el `Consumidor` debe esperar; cuando `celdasOcupadas` es `3` (el tamaño del búfer circular), el búfer circular está lleno y el `Productor` debe esperar. La variable `indiceEscritura` (línea 8) indica la siguiente ubicación en la que un `Productor` puede colocar un valor. La variable `indiceLectura` (línea 9) indica la posición a partir de la cual un `Consumidor` puede leer el siguiente valor.

El método `establecer` de `BuferCircular` (líneas 12 a 30) realiza las mismas tareas que en la figura 23.18, con unas cuantas modificaciones. El ciclo en las líneas 16 a 20 determina si el `Productor` debe esperar (es decir, todos los búferes están llenos). De ser así, en la línea 18 se indica que el `Productor` está esperando para realizar su tarea. Después, en la línea 19 se invoca el método `wait`, lo cual hace que el `Productor` libere el bloqueo de `BuferCircular` y espere hasta que haya espacio para escribir un nuevo valor en el búfer. Cuando la ejecución continúa en la línea 22 después del ciclo `while`, el valor escrito por el `Productor` se coloca en el búfer circular, en la ubicación `indiceEscritura`. Después, en la línea 25 se actualiza `indiceEscritura` para la siguiente llamada al método `establecer` de `BuferCircular`. Esta línea es la clave para la "circularidad" del búfer. Cuando `indiceEscritura` se incrementa más allá del final del búfer, la línea lo establece en `0`. En la línea 27 se incrementa `celdasOcupadas`, debido a que ahora hay un valor más en el búfer que el `Consumidor` puede leer. A continuación, en la línea 28 se invoca el método `mostrarEstado` (líneas 56 a 85) para actualizar la salida con el valor producido, el número de búferes ocupados, el contenido de esos búferes y los valores actuales de `indiceEscritura` e `indiceLectura`. En la línea 29 se invoca el método `notifyAll` para cambiar los subprocesos en espera al estado *ejecutable*, de manera que un subproceso `Consumidor` en espera (si hay uno) pueda intentar leer nuevamente un valor del búfer.

El método `obtener` de `BuferCircular` (líneas 33 a 53) también realiza las mismas tareas que hizo en la figura 23.18, con unas cuantas modificaciones menores. El ciclo en las líneas 37 a 41 determina si el `Consumidor` debe esperar (es decir, todas las celdas del búfer están vacías). Si el `Consumidor` debe esperar, en la línea 39 se actualiza la salida para indicar que el `Consumidor` está esperando realizar su tarea. Después, en la línea 40 se

invoca el método wait, lo cual hace que el subproceso actual libere el bloqueo sobre el BuferCircular y espera hasta que haya datos disponibles para leer. Cuando la ejecución continúa en un momento dado en la línea 43, después de que el Productor llama a notifyAll, a valorLeido se le asigna el valor en la ubicación indiceLectura en el búfer circular. Después, en la línea 46 se actualiza indiceLectura para la siguiente llamada al método obtener de BuferCircular. En esta línea y en la línea 25 se implementa la "circularidad" del búfer. En la línea 48 se decrementa celdasOcupadas, debido a que ahora hay una posición más en el búfer en la que el subproceso Productor puede colocar un valor. En la línea 49 se invoca el método mostrarEstado para actualizar la salida con el valor consumido, el número de búferes ocupados, el contenido de los búferes y los valores actuales de indiceEscritura e indiceLectura. En la línea 50 se invoca el método notifyAll para permitir que cualquier subproceso Productor que esté esperando escribir en el objeto BuferCircular intente escribir de nuevo. Después, en la línea 52 se devuelve el valor consumido al método que hizo la llamada.

El método mostrarEstado (líneas 56 a 85) imprime en pantalla el estado de la aplicación. En las líneas 62 y 63 se muestran los valores actuales de las celdas del búfer. En la línea 63 se utiliza el método printf con un especificador de formato "%2d" para imprimir el contenido de cada búfer con un espacio a la izquierda, si es un solo dígito. En las líneas 70 a 82 se imprimen en pantalla los valores actuales de indiceEscritura e indiceLectura con las letras W y R, respectivamente.

Prueba de la clase **BuferCircular**

La clase PruebaBuferCircular (figura 23.21) contiene el método main que inicia la aplicación. En la línea 11 se crea el objeto ExecutorService, y en la línea 14 se crea un objeto BuferCircular y se asigna su referencia a la variable BuferCircular llamada ubicacionCompartida. En la línea 17 se invoca el método mostrarEstado de BuferCircular para mostrar el estado inicial del búfer. En las líneas 20 y 21 se ejecutan las tareas Productor y Consumidor. En la línea 23 se hace una llamada al método shutdown para terminar la aplicación cuando los subprocesos completen las tareas Productor y Consumidor.

Cada vez que el Productor escribe un valor o el Consumidor lee un valor, el programa imprime en pantalla un mensaje indicando la acción realizada (lectura o escritura), el contenido de bufer y la ubicación de indiceEscritura e indiceLectura. En la salida de la figura 23.21, el Productor escribe primero el valor 1. Después, el búfer contiene el valor 1 en la primera celda y el valor -1 (el valor predeterminado que utilizamos para fines de mostrar los resultados) en las otras dos celdas. El índice de escritura se actualiza a la segunda celda, mientras que el índice de lectura permanece en la primera celda. A continuación, el Consumidor lee 1. El búfer contiene los mismos valores, pero el índice de lectura se ha actualizado a la segunda celda. Después el Consumidor trata de leer otra vez, pero el búfer está vacío y el Consumidor se ve obligado a esperar. Observe que sólo una vez en esta ejecución del programa fue necesario que uno de los dos subprocesos esperara.

```
1  // Fig 23.21: PruebaBuferCircular.java
2  // Muestra dos subprocesos que manipulan un búfer circular.
3  import java.util.concurrent.ExecutorService;
4  import java.util.concurrent.Executors;
5
6  public class PruebaBuferCircular
7  {
8     public static void main( String[] args )
9     {
10       // crea nueva reserva con dos subprocesos
11       ExecutorService aplicacion = Executors.newCachedThreadPool();
12
13       // crea objeto BuferCircular para almacenar valores int
14       BuferCircular ubicacionCompartida = new BuferCircular();
15
16       // muestra el estado inicial del objeto BuferCircular
17       ubicacionCompartida.mostrarEstado( "Estado inicial" );
```

Figura 23.21 | La aplicación muestra cómo los subprocesos Productor y Consumidor manipulan un búfer circular. (Parte 1 de 3).

```

        // ejecuta las tareas Productor y Consumidor
        aplicacion.execute( new Productor( ubicacionCompartida ) );
        aplicacion.execute( new Consumidor( ubicacionCompartida ) );

        aplicacion.shutdown();
    } // fin de main
} // fin de la clase PruebaBuferCircular
```

```
Estado inicial (celdas ocupadas del bufer: 0)
celdas bufer:   -1   -1   -1
               ---- ---- ----
                WR

Productor escribe 1 (celdas ocupadas del bufer: 1)
celdas bufer:    1   -1   -1
               ---- ---- ----
                 R   W

Consumidor lee 1 (celdas ocupadas del bufer: 0)
celdas bufer:    1   -1   -1
               ---- ---- ----
                     WR

Bufer esta vacio. Consumidor espera.
Productor escribe 2 (celdas ocupadas del bufer: 1)
celdas bufer:    1    2   -1
               ---- ---- ----
                      R   W

Consumidor lee 2 (celdas ocupadas del bufer: 0)
celdas bufer:    1    2   -1
               ---- ---- ----
                          WR

Productor escribe 3 (celdas ocupadas del bufer: 1)
celdas bufer:    1    2    3
               ---- ---- ----
                W          R

Consumidor lee 3 (celdas ocupadas del bufer: 0)
celdas bufer:    1    2    3
               ---- ---- ----
                WR

Productor escribe 4 (celdas ocupadas del bufer: 1)
celdas bufer:    4    2    3
               ---- ---- ----
                 R   W

Productor escribe 5 (celdas ocupadas del bufer: 2)
celdas bufer:    4    5    3
               ---- ---- ----
                 R        W

Consumidor lee 4 (celdas ocupadas del bufer: 1)
celdas bufer:    4    5    3
               ---- ---- ----
                      R   W
```

Figura 23.21 | La aplicación muestra cómo los subprocesos `Productor` y `Consumidor` manipulan un búfer circular. (Parte 2 de 3).

```
Productor escribe 6 (celdas ocupadas del bufer: 2)
celdas bufer:    4    5    6
               ---- ---- ----
                W     R

Productor escribe 7 (celdas ocupadas del bufer: 3)
celdas bufer:    7    5    6
               ---- ---- ----
                     WR

Consumidor lee 5 (celdas ocupadas del bufer: 2)
celdas bufer:    7    5    6
               ---- ---- ----
                     W     R

Productor escribe 8 (celdas ocupadas del bufer: 3)
celdas bufer:    7    8    6
               ---- ---- ----
                          WR

Consumidor lee 6 (celdas ocupadas del bufer: 2)
celdas bufer:    7    8    6
               ---- ---- ----
                 R        W

Consumidor lee 7 (celdas ocupadas del bufer: 1)
celdas bufer:    7    8    6
               ---- ---- ----
                      R   W

Productor escribe 9 (celdas ocupadas del bufer: 2)
celdas bufer:    7    8    9
               ---- ---- ----
                W     R

Consumidor lee 8 (celdas ocupadas del bufer: 1)
celdas bufer:    7    8    9
               ---- ---- ----
                W          R

Consumidor lee 9 (celdas ocupadas del bufer: 0)
celdas bufer:    7    8    9
               ---- ---- ----
                WR

Productor escribe 10 (celdas ocupadas del bufer: 1)
celdas bufer:   10    8    9
               ---- ---- ----
                 R   W

Productor termino de producir
Terminando Productor
Consumidor lee 10 (celdas ocupadas del bufer: 0)
celdas bufer:   10    8    9
               ---- ---- ----
                     WR

Consumidor leyo valores, el total es 55
Terminando Consumidor
```

Figura 23.21 | La aplicación muestra cómo los subprocesos `Productor` y `Consumidor` manipulan un búfer circular. (Parte 3 de 3).

23.10 Relación productor/consumidor: las interfaces Lock y Condition

Aunque la palabra clave `synchronized` proporciona la mayoría de las necesidades de sincronización de subprocesos, Java cuenta con otras herramientas para ayudar en el desarrollo de programas concurrentes. En esta sección, hablaremos sobre las interfaces `Lock` y `Condition`, que se introdujeron en Java SE 5. Estas interfaces proporcionan a los programadores un control más preciso sobre la sincronización de subprocesos, pero su uso es más complicado.

La interfaz `Lock` *y la clase* `ReentrantLock`

Cualquier objeto puede contener una referencia a un objeto que implemente a la interfaz `Lock` (del paquete `java.util.concurrent.locks`). Un subproceso llama al método `lock` de `Lock` para adquirir el bloqueo. Una vez que un subproceso obtiene un objeto `Lock`, este objeto no permitirá que otro subproceso obtenga el `Lock` sino hasta que el primer subproceso lo libere (llamando al método **`unlock`** de `Lock`). Si varios subprocesos tratan de llamar al método `lock` en el mismo objeto `Lock` y al mismo tiempo, sólo uno de estos subprocesos puede obtener el bloqueo; todos los demás se colocan en el estado *en espera* de ese bloqueo. Cuando un subproceso llama al método `unlock`, se libera el bloqueo sobre el objeto y un subproceso en espera que intente bloquear el objeto puede continuar.

La clase **`ReentrantLock`** (del paquete `java.util.concurrent.locks`) es una implementación básica de la interfaz `Lock`. El constructor de `ReentrantLock` recibe un argumento `boolean`, el cual especifica si el bloqueo tiene una política de equidad. Si el argumento es `true`, la **política de equidad** de `ReentrantLock` es: "el subproceso con más tiempo de espera adquirirá el bloqueo cuando esté disponible". Dicha política de equidad garantiza que nunca ocurra el aplazamiento indefinido (también conocido como inanición). Si el argumento de la política de equidad se establece en `false`, no hay garantía en cuanto a cuál subproceso en espera adquirirá el bloqueo cuando esté disponible.



Observación de ingeniería de software 23.2

El uso de un objeto `ReentrantLock` *con una política de equidad evita el aplazamiento indefinido.*



Tip de rendimiento 23.4

El uso de un objeto `ReentrantLock` *con una política de equidad puede reducir el rendimiento del programa, de una manera considerable.*

Los objetos de condición y la interfaz `Condition`

Si un subproceso que posee un objeto `Lock` determina que no puede continuar con su tarea hasta que se cumpla cierta condición, el subproceso puede esperar en base a un **objeto de condición**. El uso de objetos `Lock` nos permite declarar de manera explícita los objetos de condición sobre los cuales un subproceso tal vez tenga que esperar. Por ejemplo, en la relación productor/consumidor los productores pueden esperar en base a un objeto, y los consumidores en base a otro. Esto no es posible cuando se utiliza la palabra clave `synchronized` y el bloqueo de monitor integrado de un objeto. Los objetos de condición se asocian con un objeto `Lock` específico y se crean mediante una llamada al método **`newCondition`** de Lock, el cual devuelve un objeto que implementa a la interfaz **`Condition`** (del paquete `java.util.concurrent.locks`). Para esperar en base a un objeto de condición, el subproceso puede llamar al método **`await`** de `Condition`. Esto libera de inmediato el objeto `Lock` asociado, y coloca al subproceso en el estado *en espera*, en base a ese objeto `Condition`. Así, otros subprocesos pueden tratar de obtener el objeto `Lock`. Cuando un subproceso *ejecutable* completa una tarea y determina que el subproceso *en espera* puede ahora continuar, el subproceso *ejecutable* puede llamar al método **`signal`** de `Condition` para permitir que un subproceso en el estado *en espera* de ese objeto `Condition` regrese al estado *ejecutable*. En este punto, el subproceso que cambió del estado *en espera* al estado *ejecutable* puede tratar de readquirir el objeto `Lock`. Aun si puede readquirir el objeto `Lock`, tal vez el subproceso no pueda todavía realizar su tarea en este momento; en cuyo caso, el subproceso puede llamar al método `await` de `Condition` para liberar el objeto `Lock` y volver a entrar al estado *en espera*. Si hay varios subprocesos en un estado *en espera* de un objeto `Condition` cuando se hace la llamada a `signal`, la implementación predeterminada de `Condition` indica al subproceso con más tiempo de espera que debe cambiar al estado *ejecutable*. Si un subproceso llama al método **`signalAll`** de `Condition`,

entonces todos los subprocesos que esperan esa condición cambian al estado *ejecutable* y se convierten en candidatos para readquirir el objeto `Lock`. Sólo uno de esos subprocesos puede obtener el bloqueo (`Lock`) sobre el objeto; los demás esperarán hasta que el objeto `Lock` esté disponible otra vez. Si el objeto `Lock` tiene una política de equidad, el subproceso con más tiempo de espera adquiere ese objeto `Lock`. Cuando un subproceso termina con un objeto compartido, debe llamar al método `unlock` para liberar al objeto `Lock`.



Error común de programación 23.2

*El **interbloqueo** (deadlock) ocurre cuando un subproceso en espera (al cual llamaremos subproceso1) no puede continuar, debido a que está esperando (ya sea en forma directa o indirecta) a que otro subproceso (al cual llamaremos subproceso2) continúe, mientras que al mismo tiempo, el subproceso2 no puede continuar debido a que está esperando (ya sea en forma directa o indirecta) a que el subproceso 1 continúe. Los dos subprocesos están en espera uno del otro, por lo que las acciones que permitirían a cada subproceso continuar su ejecución nunca ocurrirán.*



Tip para prevenir errores 23.3

Cuando varios subprocesos manipulan a un objeto compartido mediante el uso de bloqueos, debemos asegurarnos que, si un subproceso llama al método `await` *para entrar al estado en espera de un objeto de condición, en algún momento dado un subproceso separado llamará al método* `signal` *de* `Condition` *para cambiar el subproceso que espera al objeto de condición de vuelta al estado* ejecutable. *Si puede haber varios subprocesos esperando el objeto de condición, un subproceso separado puede llamar al método* `signalAll` *de* `Condition` *como garantía para asegurar que todos los subprocesos en espera tengan otra oportunidad para realizar sus tareas. Si esto no se hace, puede ocurrir un aplazamiento indefinido (inanición).*



Error común de programación 23.3

Una excepción `IllegalMonitorStateException` *ocurre si un subproceso llama a* `await`, `signal` *o* `signalAll` *en base a un objeto de condición, sin haber adquirido el bloqueo para ese objeto de condición.*

Comparación entre `Lock` *y* `Condition`*, y la palabra clave* `synchronized`

En algunas aplicaciones, el uso de objetos `Lock` y `Condition` puede ser preferible a utilizar la palabra clave `synchronized`. Los objetos `Lock` nos permiten interrumpir a los subprocesos en espera, o especificar un tiempo límite para esperar a adquirir un bloqueo, lo cual no es posible si se utiliza la palabra clave `synchronized`. Además, un objeto `Lock` no está restringido a ser adquirido y liberado en el mismo bloque de código, lo cual es el caso con la palabra clave `synchronized`. Los objetos `Condition` nos permiten especificar varios objetos de condición, en base a los cuales los subprocesos pueden esperar. Por ende, es posible indicar a los subprocesos en espera que un objeto de condición específico es ahora verdadero, llamando a `signal` o `signalAll` en ese objeto `Condition`. Con la palabra clave `synchronized`, no hay forma de indicar de manera explícita la condición en la cual esperan los subprocesos y, por lo tanto, no hay forma de notificar a los subprocesos en espera de una condición específica que pueden continuar, sin también indicarlo a los subprocesos que están en espera de otras condiciones. Hay otras posibles ventajas en cuanto al uso de objetos `Lock` y `Condition`, pero debemos tener en cuenta que, por lo general, es mejor utilizar la palabra clave `synchronized`, a menos que nuestra aplicación requiera capacidades avanzadas de sincronización. El uso de las interfaces `Lock` y `Condition` es propenso a errores; no se garantiza la llamada a `unlock`, mientras que el monitor en una instrucción `synchronized` siempre se liberará cuando la instrucción termine de ejecutarse.

Uso de objetos `Lock` *y* `Condition` *para implementar la* `sincronización`

Para ilustrar cómo usar las interfaces `Lock` y `Condition`, ahora vamos a implementar la relación productor/consumidor, utilizando objetos `Lock` y `Condition` para coordinar el acceso a un búfer compartido con un solo elemento (figuras 23.22 y 23.23). En este caso, cada valor producido se consume correctamente sólo una vez.

La clase `BuferSincronizado` (figura 23.22) contiene cinco campos. En la línea 11 se crea un nuevo objeto de tipo `ReentrantLock` y se asigna su referencia a la variable `Lock` llamada `bloqueoAcceso`. El objeto `bloqueoReentrante` se crea sin la política de equidad, ya que en cualquier momento dado, sólo un `Productor` o `Consumidor` estará esperando adquirir el objeto `Lock` en este ejemplo. En las líneas 14 y 15 se crean dos objetos `Condition` mediante el uso del método `newCondition` de `Lock`. El objeto `Condition` llamado `puedeEscribir` contiene una cola para un subproceso `Productor`, el cual espera mientras el búfer esté lleno (es decir, hay datos

en el búfer pero el `Consumidor` no los ha leído aún). Si el búfer está lleno, el `Productor` llama al método `await` en este objeto `Condition`. Cuando el `Consumidor` lee datos de un búfer lleno, llama al método `signal` en este objeto `Condition`. El objeto `Condition puedeLeer` contiene una cola para un `Consumidor` que espera mientras el búfer esté vacío (es decir, no hay datos en el búfer para que el `Consumidor` los lea). Si el búfer está vacío, el `Consumidor` llama al método `await` en este objeto `Condition`. Cuando el `Productor` escribe en el búfer vacío, llama al método `signal` en este objeto `Condition`. La variable `int bufer` (línea 17) contiene los datos compartidos. La variable `boolean ocupado` (línea 18) mantiene el rastro acerca de si el búfer contiene datos en un momento dado (que el `Consumidor` debe leer).

```
1   // Fig. 23.22: BuferSincronizado.java
2   // Sincroniza el acceso a un entero compartido, usando las interfaces
3   // Lock y Condition
4   import java.util.concurrent.locks.Lock;
5   import java.util.concurrent.locks.ReentrantLock;
6   import java.util.concurrent.locks.Condition;
7
8   public class BuferSincronizado implements Bufer
9   {
10     // Bloqueo para controlar la sincronización con este búfer
11     private final Lock bloqueoAcceso = new ReentrantLock();
12
13     // condiciones para controlar la lectura y escritura
14     private final Condition puedeEscribir = bloqueoAcceso.newCondition();
15     private final Condition puedeLeer = bloqueoAcceso.newCondition();
16
17     private int bufer = -1; // compartido por los subprocesos productor y consumidor
18     private boolean ocupado = false; // indica si el búfer está ocupado
19
20     // coloca un valor int en el búfer
21     public void establecer( int valor ) throws InterruptedException
22     {
23        bloqueoAcceso.lock(); // bloquea este objeto
24
25        // imprime información del subproceso y del búfer, después espera
26        try
27        {
28           // mientras búfer no esté vacío, coloca el subproceso en espera
29           while ( ocupado )
30           {
31              System.out.println( "Productor trata de escribir.");
32              mostrarEstado( "Bufer lleno. Productor espera." );
33              puedeEscribir.await(); // espera hasta que bufer esté vacío
34           } // fin de while
35
36           bufer = valor; // establece el nuevo valor de búfer
37
38           // indica que el productor no puede almacenar otro valor
39           // hasta que el consumidor obtenga el valor actual del búfer
40           ocupado = true;
41
42           mostrarEstado( "Productor escribe " + búfer );
43
44           // indica al subproceso en espera que lea del búfer
45           puedeLeer.signal();
46        } // fin de try
47        finally
48        {
```

Figura 23.22 | Sincroniza el acceso a un entero compartido, usando las interfaces `Lock` y `Condition`. (Parte 1 de 2).

```
49             bloqueoAcceso.unlock(); // desbloquea este objeto
50          } // fin de finally
51       } // fin del método establecer
52
53       // devuelve el valor del búfer
54       public int obtener() throws InterruptedException
55       {
56          int valorLeido = 0; // inicializa el valor que se leyó del búfer
57          bloqueoAcceso.lock(); // bloquea este objeto
58
59          // imprime información del subproceso y del búfer, después espera
60          try
61          {
62             // mientras no haya datos qué leer, coloca el subproceso en espera
63             while ( !ocupado )
64             {
65                System.out.println( "Consumidor trata de leer." );
66                mostrarEstado( "Bufer vacio. Consumidor espera." );
67                puedeLeer.await(); // espera hasta que bufer esté lleno
68             } // fin de while
69
70             // indica que el productor puede almacenar otro valor
71             // porque el consumidor acaba de obtener el valor del búfer
72             ocupado = false;
73
74             valorLeido = bufer; // obtiene el valor del búfer
75             mostrarEstado( "Consumidor lee " + valorLeido );
76
77             // indica al subproceso que espera a que el búfer esté vacío
78             puedeEscribir.signal();
79          } // fin de try
80          finally
81          {
82             bloqueoAcceso.unlock(); // desbloquea este objeto
83          } // fin de finally
84
85          return valorLeido;
86       } // fin del método obtener
87
88       // muestra la operación actual y el estado del búfer
89       public void mostrarEstado( String operacion )
90       {
91          System.out.printf( "%-40s%d\t\t%b\n\n", operacion, bufer,
92             ocupado );
93       } // fin del método mostrarEstado
94    } // fin de la clase BuferSincronizado
```

Figura 23.22 | Sincroniza el acceso a un entero compartido, usando las interfaces `Lock` y `Condition`. (Parte 2 de 2).

En la línea 23, en el método `establecer` se hace una llamada al método `lock` en el objeto `bloqueoAcceso` de `BuferSincronizado`. Si el bloqueo está disponible (es decir, si ningún otro subproceso ha adquirido este bloqueo), el método `lock` regresa de inmediato (este subproceso ahora posee el bloqueo) y el subproceso continúa. Si el bloqueo no está disponible (es decir, si lo posee otro subproceso), este método espera hasta que el otro subproceso libere el bloqueo. Una vez que se adquiere el bloqueo, se ejecuta el bloque `try` en las líneas 26 a 46. En la línea 29 se evalúa `ocupado` para determinar si `bufer` está lleno. Si es así, en las líneas 31 y 32 se muestra un mensaje indicando que el subproceso va a esperar. En la línea 33 se hace una llamada al método `await` de `Condition` en el objeto de condición `puedeEscribir`, lo cual libera temporalmente el objeto `Lock` de `BuferSincronizado` y espera una señal del `Consumidor`, indicando que el `bufer` está disponible para escritura. Cuando `bufer` está

disponible, el método continúa y escribe en `bufer` (línea 36), establece `ocupado` en `true` (línea 40) y muestra un mensaje indicando que el productor escribió un valor (línea 42). En la línea 45 se hace una llamada al método `signal` de `Condition` en el objeto de condición `puedeLeer`, para notificar al `Consumidor` en espera (si hay uno) que el búfer tiene nuevos datos para leer. En la línea 49 se hace una llamada al método `unlock` desde un bloque `finally` para liberar el bloqueo y permitir que el `Consumidor` continúe.



Tip para prevenir errores 23.4

Coloque las llamadas al método `unlock` *de Lock en un bloque* `finally`*. Si se lanza una excepción,* `unlock` *debe llamarse de todas formas, o de lo contrario ocurriría un interbloqueo.*

En la línea 57 del método `obtener` (líneas 54 a 86) se hace una llamada al método `lock` para adquirir el objeto `Lock`. Este método espera hasta que el objeto `Lock` esté disponible. Una vez que se adquiere este objeto `Lock`, en la línea 63 se evalúa si ocupado es `false`, lo cual indica que el búfer está vacío. Si es así, en la línea 67 se hace una llamada al método `await` en el objeto de condición `puedeLeer`. Recuerde que el método `signal` se llama en la variable `puedeLeer`, en el método `establecer` (línea 45). Cuando se hace una indicación al objeto `Condition`, el método `obtener` continúa. En la línea 72 se establece `ocupado` en `false`, en la línea 74 se almacena el valor de `bufer` en `valorLeido` y en la línea 75 se imprime en pantalla el `valorLeido`. Después, en la línea 78 se hace una indicación al objeto de condición `puedeEscribir`. Esto despertará al `Productor` si es que está esperando a que el búfer esté vacío. En la línea 82 se hace una llamada al método `unlock` desde un bloque `finally` para liberar el bloqueo, y en la línea 85 se devuelve `valorLeido` al método que hizo la llamada.



Error común de programación 23.4

Olvidar hacer una indicación mediante `signal` *a un subproceso en espera es un error lógico. El subproceso permanecerá en el estado* en espera*, lo cual evitará que pueda continuar. Dicha espera puede producir un aplazamiento indefinido o un interbloqueo.*

La clase `PruebaBuferCompartido2` (figura 23.23) es idéntica a la de la figura 23.19. Estudie el conjunto de resultados de la figura 23.23. Observe que cada entero producido se consume exactamente una vez; no se pierden valores, y no se consumen valores más de una vez. Los objetos `Lock` y `Condition` aseguran que el `Productor` y el `Consumidor` no puedan realizar sus tareas, a menos que sea su turno. El `Productor` debe ir primero, el `Consumidor` debe esperar si el `Productor` no ha producido desde la última vez que el `Consumidor` consumió, y el `Productor` debe esperar si el `Consumidor` no ha consumido aún el valor que el `Productor` produjo más recientemente. Ejecute este programa varias veces para confirmar que cada entero producido sea consumido exactamente una vez. En los resultados de ejemplo observe las líneas resaltadas, las cuales indican cuándo deben esperar el `Productor` y el `Consumidor` para realizar sus respectivas tareas.

```java
// Fig 23.23: PruebaBuferCompartido2.java
// Dos subprocesos que manipulan un búfer sincronizado.
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class PruebaBuferCompartido2
{
   public static void main( String[] args )
   {
      // crea nueva reserva con dos subprocesos
      ExecutorService aplicacion = Executors.newCachedThreadPool();

      // crea BuferSincronizado para almacenar valores int
      Bufer ubicacionCompartida = new BuferSincronizado();

      System.out.printf( "40s%s\t\t%s\n%-40s%s\n\n", "Operacion",
```

Figura 23.23 | Dos subprocesos que manipulan un búfer sincronizado. (Parte 1 de 3).

```
            "Bufer", "Ocupado", "---------", "------\t\t--------" );

         // ejecuta las tareas Productor y Consumidor
         aplicacion.execute( new Productor( ubicacionCompartida ) );
         aplicacion.execute( new Consumidor( ubicacionCompartida ) );

         aplicacion.shutdown();
      } // fin de main
   } // fin de la clase PruebaBuferCompartido2
```

```
Operacion                                    Bufer          Ocupado
---------                                    ------         --------

Productor escribe 1                          1              true

Productor trata de escribir.
Bufer lleno. Productor espera.               1              true


Consumidor lee 1                             1              false

Productor escribe 2                          2              true

Productor trata de escribir.
Bufer lleno. Productor espera.               2              true

Consumidor lee 2                             2              false

Productor escribe 3                          3              true

Consumidor lee 3                             3              false

Productor escribe 4                          4              true

Consumidor lee 4                             4              false

Consumidor trata de leer.
Bufer vacio. Consumidor espera.              4              false

Productor escribe 5                          5              true

Consumidor lee 5                             5              false

Consumidor trata de leer.
Bufer vacio. Consumidor espera.              5              false

Productor escribe 6                          6              true

Consumidor lee 6                             6              false

Productor escribe 7                          7              true

Consumidor lee 7                             7              false

Productor escribe 8                          8              true

Consumidor lee 8                             8              false

Productor escribe 9                          9              true
```

Figura 23.23 | Dos subprocesos que manipulan un búfer sincronizado. (Parte 2 de 3).

```
Consumidor lee 9                          9          false

Productor escribe 10                      10         true


Productor termino de producir
Terminando Productor
Consumidor lee 10                         10         false


Consumidor leyo valores, el total es 55
Terminando Consumidor
```

Figura 23.23 | Dos subprocesos que manipulan un búfer sincronizado. (Parte 3 de 3).

23.11 Subprocesamiento múltiple con GUIs

Las aplicaciones de Swing presentan un conjunto único de retos para la programación con subprocesamiento múltiple. Todas las aplicaciones de Swing tienen un solo subproceso, conocido como el **subproceso de despachamiento de eventos**, para manejar las interacciones con los componentes de la GUI de la aplicación. Las interacciones típicas incluyen actualizar los componentes de la GUI o procesar las acciones del usuario, como los clics del ratón. Todas las tareas que requieren interacción con la GUI de una aplicación se colocan en una cola de eventos y se ejecutan en forma secuencial, mediante el subproceso de despachamiento de eventos.

Los componentes de GUI de Swing no son seguros para los subprocesos; no se pueden manipular mediante varios subprocesos sin el riesgo de obtener resultados incorrectos. A diferencia de los demás ejemplos que se presentan en este capítulo, la seguridad de subprocesos en aplicaciones de GUI se logra, no mediante la sincronización de las acciones de los subprocesos, sino asegurando que se acceda a los componentes de Swing sólo desde un solo subproceso: el subproceso de despachamiento de eventos. A esta técnica se le conoce como **confinamiento de subprocesos**. Al permitir que sólo un subproceso acceda a los objetos que no son seguros para los subprocesos, se elimina la posibilidad de corrupción debido a que varios subprocesos accedan a estos objetos en forma concurrente.

Por lo general, basta con realizar cálculos simples en el subproceso de despachamiento de eventos, en secuencia con las manipulaciones de los componentes de GUI. Si una aplicación debe realizar un cálculo extenso en respuesta a una interacción con la interfaz del usuario, el subproceso de despachamiento de eventos no puede atender otras tareas en la cola de eventos, mientras se encuentre atado en ese cálculo. Esto hace que los componentes de la GUI pierdan su capacidad de respuesta. Es preferible manejar un cálculo extenso en un subproceso separado, con lo cual el subproceso de despachamiento de eventos queda libre para continuar administrando las demás interacciones con la GUI. Desde luego que, para actualizar la GUI con base en los resultados del cálculo, debemos actualizar la GUI desde el subproceso de despachamiento de eventos, en vez de hacerlo desde el subproceso trabajador que realizó el cálculo.

La clase `SwingWorker`

Java SE 6 cuenta con la clase **`SwingWorker`** (en el paquete `javax.swing`) para realizar cálculos extensos en un subproceso trabajador, y para actualizar los componentes de Swing desde el subproceso de despachamiento de eventos, con base en los resultados del cálculo. `SwingWorker` implementa a la interfaz `Runnable`, lo cual significa que un objeto `SwingWorker` se puede programar para ejecutarse en un subproceso separado. La clase `SwingWorker` proporciona varios métodos para simplificar la realización de cálculos en un subproceso trabajador, y hacer que estos resultados estén disponibles para mostrarlos en una GUI. En la figura 23.24 se describen algunos métodos comunes de `SwingWorker`.

23.11.1 Realización de cálculos en un subproceso trabajador

En el siguiente ejemplo, una GUI proporciona componentes para que el usuario escriba un número *n* y obtenga el *n*-ésimo número de Fibonacci, el cual calculamos mediante el uso del algoritmo recursivo que vimos en la sección 15.4. Como el algoritmo recursivo consume mucho tiempo para valores extensos, utilizamos un objeto

Método	Descripción
`doInBackground`	Define un cálculo extenso y se llama desde un subproceso trabajador.
`done`	Se ejecuta en el subproceso de despachamiento de eventos, cuando `doInBackground` regresa.
`execute`	Programa el objeto `SwingWorker` para que se ejecute en un subproceso trabajador.
`get`	Espera a que se complete el cálculo, y después devuelve el resultado del mismo (es decir, el valor de retorno de `doInBackground`).
`publish`	Envía resultados inmediatos del método `doInBackground` al método `process`, para procesarlos en el subproceso de despachamiento de eventos.
`process`	Recibe los resultados intermedios del método `publish` y los procesa en el subproceso de despachamiento de eventos.
`setProgress`	Establece la propiedad de progreso para notificar a cualquier componente de escucha de cambio de propiedades que esté en el subproceso de despachamiento de eventos, acerca de las actualizaciones en la barra de progreso.

Figura 23.24 | Métodos de uso común de `SwingWorker`.

`SwingWorker` para realizar el cálculo en un subproceso trabajador. La GUI también proporciona un conjunto separado de componentes que obtienen el siguiente número de Fibonacci en secuencia con cada clic de un botón, empezando con `fibonacci(1)`. Este conjunto de componentes realiza su cálculo corto directamente en el subproceso de despachamiento de eventos.

La clase `CalculadoraSegundoPlano` (figura 23.25) realiza el cálculo recursivo de Fibonacci en un subproceso trabajador. Esta clase extiende a `SwingWorker` (línea 8), sobrescribiendo a los métodos `doInBackground` y `done`. El método `doInBackground` (líneas 21 a 25) calcula el *n*-ésimo número de Fibonacci en un subproceso trabajador y devuelve el resultado. El método `done` (líneas 28 a 44) muestra el resultado en un objeto `JLabel`.

```
1  // Fig. 23.25: CalculadoraSegundoPlano.java
2  // Subclase de SwingWorker para calcular números de Fibonacci
3  // en un subproceso en segundo plano.
4  import javax.swing.SwingWorker;
5  import javax.swing.JLabel;
6  import java.util.concurrent.ExecutionException;
7
8  public class CalculadoraSegundoPlano extends SwingWorker< String, Object >
9  {
10    private final int n; // número de Fibonacci a calcular
11    private final JLabel resultadoJLabel; // JLabel para mostrar el resultado
12
13    // constructor
14    public CalculadoraSegundoPlano( int numero, JLabel etiqueta )
15    {
16       n = numero;
17       resultadoJLabel = etiqueta;
18    } // fin del constructor de CalculadoraSegundoPlano
19
20    // código que tarda mucho en ejecutarse, para ejecutarlo en un subproceso trabajador
21    public String doInBackground()
22    {
23       long nesimoFib = fibonacci( n );
```

Figura 23.25 | Subclase de `SwingWorker` para calcular números de Fibonacci en un subproceso en segundo plano. (Parte 1 de 2).

```
24          return String.valueOf( nesimoFib );
25       } // fin del método doInBackground
26
27       // código para ejecutar en el subproceso de despachamiento de eventos cuando regresa
         doInBackground
28       protected void done()
29       {
30          try
31          {
32             // obtiene el resultado de doInBackground y lo muestra
33             resultadoJLabel.setText( get() );
34          } // fin de try
35          catch ( InterruptedException ex )
36          {
37             resultadoJLabel.setText( "Se interrumpio mientras esperaba los resultados." );
38          } // fin de catch
39          catch ( ExecutionException ex )
40          {
41             resultadoJLabel.setText(
42                "Se encontro un error al realizar el calculo." );
43          } // fin de catch
44       } // fin del método done
45
46       // método recursivo fibonacci; calcula el n-ésimo número de Fibonacci
47       public long fibonacci( long numero )
48       {
49          if ( numero == 0 || numero == 1 )
50             return numero;
51          else
52             return fibonacci( numero - 1 ) + fibonacci( numero - 2 );
53       } // fin del método fibonacci
54    } // fin de la clase CalculadoraSegundoPlano
```

Figura 23.25 | Subclase de `SwingWorker` para calcular números de Fibonacci en un subproceso en segundo plano. (Parte 2 de 2).

Observe que `SwingWorker` es una clase genérica. En la línea 8, el primer parámetro de tipo es `String` y el segundo es `Object`. El primer parámetro de tipo indica el tipo devuelto por el método `doInBackground`; el segundo indica el tipo que se pasa entre los métodos `publish` y `process` para manejar los resultados intermedios. Como no utilizamos a `publish` o a `process` en este ejemplo, simplemente usamos `Object` como el segundo parámetro de tipo. En la sección 23.11.2 hablaremos sobre `publish` y `process`.

Un objeto `CalculadoraSegundoPlano` puede instanciarse a partir de una clase que controle una GUI. Un objeto `CalculadoraSegundoPlano` mantiene variables de instancia para un entero que representa el número de Fibonacci a calcular, y un objeto `Jlabel` que muestra los resultados del cálculo (líneas 10 y 11). El constructor de `CalculadoraSegundoPlano` (líneas 14 a 18) inicializa estas variables de instancia con los argumentos que se pasan al constructor.



Observación de ingeniería de software 23.3

Cualquier componente de la GUI que se manipule mediante métodos de `SwingWorker`*, como los componentes que se actualizan a partir de los métodos* `process` *o* `done`*, se debe pasar al constructor de la subclase de* `SwingWorker` *para almacenarlo en el objeto de la subclase. Esto proporciona a estos métodos acceso a los componentes de la GUI que van a manipular.*

Cuando se hace una llamada al método `execute` en un objeto `CalculadoraSegundoPlano`, el objeto se programa para ejecutarlo en un subproceso trabajador. El método `doInBackground` se llama desde el subproceso trabajador e invoca al método `fibonacci` (líneas 47 a 53), en donde recibe la variable de instancia `n` como argumento (línea 23). El método `fibonacci` utiliza la recursividad para calcular el número de Fibonacci de `n`. Cuando `fibonacci` regresa, el método `doInBackground` devuelve el resultado.

Una vez que `doInBackground` regresa, el método `done` se llama desde el subproceso de despachamiento de eventos. Este método trata de asignar al objeto `JLabel` que contiene el resultado del valor de retorno de `doInBackground`, mediante una llamada al método `get` para obtener este valor de retorno (línea 33). El método `get` espera a que el resultado este listo, en caso de ser necesario, pero como lo llamamos desde el método `done`, el cálculo se completará antes de que se llame a `get`. En las líneas 35 a 38 se atrapa una excepción `InterruptedException` si el subproceso actual se interrumpe mientras espera a que `get` regrese. En las líneas 39 a 43 se atrapa una excepción `ExecutionException`, que se lanza si ocurre una excepción durante el cálculo.

La clase `NumerosFibonacci` (figura 23.26) muestra una ventana que contiene dos conjuntos de componentes de GUI: uno para calcular un número de Fibonacci en un subproceso trabajador, y otro para obtener el siguiente número de Fibonacci en respuesta a la acción del usuario de oprimir un botón `JButton`. El constructor (líneas 38 a 109) coloca estos componentes en objetos `JPanel` con títulos separados. En las líneas 46 a 47 y 78 a 79 se agregan dos objetos `JLabel` un objeto `JTextField` y un objeto `JButton` al objeto `trabajadorJPanel` para que el usuario pueda introducir un entero, cuyo número de Fibonacci se calculará mediante el objeto `BackgroundWorker`. En las líneas 84 a 85 y 103 se agregan dos objetos `JLabel` y un objeto `JButton` al panel del subproceso de despachamiento de eventos, para permitir al usuario obtener el siguiente número de Fibonacci en la secuencia. Las variables de instancia `n1` y `n2` contienen los dos números de Fibonacci anteriores en la secuencia, y se inicializan con `0` y `1` respectivamente (líneas 29 y 30). La variable de instancia `cuenta` almacena el número en la secuencia que se calculó más recientemente, y se inicializa con `1` (línea 31). Al principio, los dos objetos `JLabel` muestran a cuenta y a `n2`, de manera que el usuario pueda ver el texto `Fibonacci de 1: 1` en el objeto `subprocesoEventosJPanel` cuando la GUI inicia.

```
1  // Fig. 23.26: NumerosFibonacci.java
2  // Uso de SwingWorker para realizar un cálculo extenso, en donde
3  // los resultados intermedios se muestran en una GUI.
4  import java.awt.GridLayout;
5  import java.awt.event.ActionEvent;
6  import java.awt.event.ActionListener;
7  import javax.swing.JButton;
8  import javax.swing.JFrame;
9  import javax.swing.JPanel;
10 import javax.swing.JLabel;
11 import javax.swing.JTextField;
12 import javax.swing.border.TitledBorder;
13 Import javax.swing.border.LineBorder;
14 import java.awt.Color;
15 import java.util.concurrent.ExecutionException;
16
17 public class NumerosFibonacci extends JFrame
18 {
19    // componentes para calcular el valor de Fibonacci de un número introducido por el
      usuario
20    private final JPanel trabajadorJPanel =
21       new JPanel( new GridLayout( 2, 2, 5, 5 ) );
22    private final JTextField numeroJTextField = new JTextField();
23    private final JButton iniciarJButton = new JButton( "Iniciar" );
24    private final JLabel fibonacciJLabel = new JLabel();
25
26    // componentes y variables para obtener el siguiente número de Fibonacci
27    private final JPanel subprocesoEventosJPanel =
28       new JPanel( new GridLayout( 2, 2, 5, 5 ) );
29    private int n1 = 0; // se inicializa con el primer número de Fibonacci
30    private int n2 = 1; // se inicializa con el segundo número de Fibonacci
31    private int cuenta = 1;
32    private final JLabel nJLabel = new JLabel( "Fibonacci de 1: " );
```

Figura 23.26 | Uso de `SwingWorker` para realizar un cálculo extenso, en donde los resultados intermedios se muestran en una GUI. (Parte 1 de 3).

```
   private final JLabel nFibonacciJLabel =
      new JLabel( String.valueOf( n2 ) );
   private final JButton siguienteNumeroJButton = new JButton( "Siguiente numero");

   // constructor
   public NumerosFibonacci()
   {
      super( "Numeros de Fibonacci" );
      setLayout( new GridLayout( 2, 1, 10, 10 ) );

      // agrega componentes de GUI al panel de SwingWorker
      trabajadorJPanel.setBorder( new TitledBorder(
         new LineBorder( Color.BLACK ), "Con SwingWorker" ) );
      trabajadorJPanel.add( new JLabel( "Obtener Fibonacci de:" ) );
      trabajadorJPanel.add( numeroJTextField );
      iniciarJButton.addActionListener(
         new ActionListener()
         {
            public void actionPerformed( ActionEvent evento )
            {
               int n;

               try
               {
                  // obtiene la entrada del usuario como un entero
                  n = Integer.parseInt( numeroJTextField.getText() );
               } // fin de try
               catch( NumberFormatException ex )
               {
                  // muestra un mensaje de error si el usuario no
                  // introdujo un entero
                  fibonacciJLabel.setText( "Escriba un entero." );
                  return;
               } // fin de catch

               // indica que ha empezado el cálculo
               fibonacciJLabel.setText( "Calculando..." );

               // crea una tarea para realizar el cálculo en segundo plano
               CalculadoraSegundoPlano tarea =
                  new CalculadoraSegundoPlano( n, fibonacciJLabel );
               tarea.execute(); // ejecuta la tarea
            } // fin del método actionPerformed
         } // fin de la clase interna anónima
      ); // fin de la llamada a addActionListener
      trabajadorJPanel.add( iniciarJButton );
      trabajadorJPanel.add( fibonacciJLabel );

      // agrega componentes de GUI al panel del subproceso de despachamiento de eventos
      subprocesoEventosJPanel.setBorder( new TitledBorder(
         new LineBorder( Color.BLACK ), "Sin SwingWorker" ) );
      subprocesoEventosJPanel.add( nJLabel );
      subprocesoEventosJPanel.add( nFibonacciJLabel );
      siguienteNumeroJButton.addActionListener(
         new ActionListener()
         {
            public void actionPerformed( ActionEvent evento )
            {
```

Figura 23.26 | Uso de `SwingWorker` para realizar un cálculo extenso, en donde los resultados intermedios se muestran en una GUI. (Parte 2 de 3).

```
91                // calcula el número de Fibonacci después de n2
92                int temp = n1 + n2;
93                n1 = n2;
94                n2 = temp;
95                ++cuenta;
96
97                // muestra el siguiente número de Fibonacci
98                nJLabel.setText( "Fibonacci de " + cuenta + ": " );
99                nFibonacciJLabel.setText( String.valueOf( n2 ) );
100            } // fin del método actionPerformed
101         } // fin de la clase interna anónima
102      ); // fin de la llamada a addActionListener
103      subprocesoEventosJPanel.add( siguienteNumeroJButton );
104
105      add( trabajadorJPanel );
106      add( subprocesoEventosJPanel );
107      setSize( 275, 200 );
108      setVisible( true );
109   } // fin del constructor
110
111   // el método main empieza la ejecución del programa
112   public static void main( String[] args )
113   {
114      NumerosFibonacci aplicacion = new NumerosFibonacci();
115      aplicacion.setDefaultCloseOperation( EXIT_ON_CLOSE );
116   } // fin de main
117 } // fin de la clase NumerosFibonacci
```

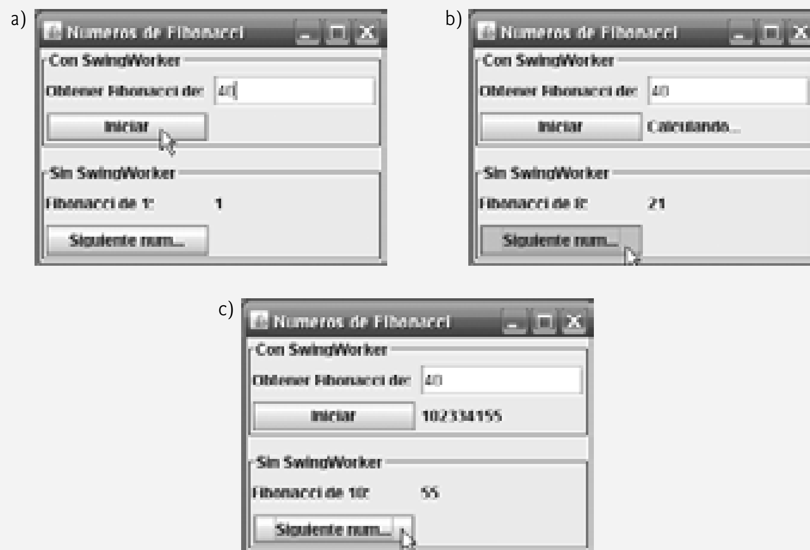


Figura 23.26 | Uso de `SwingWorker` para realizar un cálculo extenso, en donde los resultados intermedios se muestran en una GUI. (Parte 3 de 3).

En las líneas 48 a 77 se registra el manejador de eventos para el botón `iniciarJButton`. Si el usuario hace clic en este objeto `JButton`, en la línea 58 se obtiene el valor introducido en el objeto `numeroJTextField` y el programa intenta convertirlo en entero. En las líneas 72 y 73 se crea un nuevo objeto `CalculadoraSegundoPlano`, el cual recibe el valor introducido por el usuario y el objeto `fibonacciJLabel` que se utiliza para mostrar los resultados del cálculo. En la línea 74 se hace una llamada al método `execute` en el objeto `CalculadoraSegun-`

`doPlano`, y se programa para ejecutarlo en un subproceso trabajador separado. El método `execute` no espera a que el objeto `CalculadoraSegundoPlano` termine de ejecutarse. Regresa de inmediato, lo cual permite a la GUI continuar procesando otros eventos, mientras se realiza el cálculo.

Si el usuario hace clic en el objeto `siguienteNumeroJButton` que está en el objeto `subprocesoEventosJPanel`, se ejecuta el manejador de eventos registrado en las líneas 86 a 102. Los dos números de Fibonacci anteriores que están almacenados en `n1` y `n2` se suman y `cuenta` se incrementa para determinar el siguiente número en la secuencia (líneas 92 a 95). Después, en las líneas 98 y 99 se actualiza la GUI para mostrar el siguiente número. El código para realizar estos cálculos está escrito directamente en el método `actionPerformed`, por lo que estos cálculos se llevan a cabo en el subproceso de despachamiento de eventos. Al manejar estos cálculos cortos en el subproceso de despachamiento de eventos, la GUI no pierde capacidad de respuesta, como el algoritmo recursivo para calcular el valor de Fibonacci para un número extenso. Debido a que el cálculo del número de Fibonacci más extenso se realiza en un subproceso trabajador separado mediante el uso del objeto `SwingWorker`, es posible obtener el siguiente número de Fibonacci mientras el cálculo recursivo aún se está realizando.

23.11.2 Procesamiento de resultados inmediatos con SwingWorker

Hemos presentado un ejemplo en el que se utiliza la clase `SwingWorker` para ejecutar un proceso extenso en un subproceso en segundo plano, en donde la GUI se actualiza al terminar el proceso. Ahora presentaremos un ejemplo acerca de cómo actualizar la GUI con resultados intermedios antes de que el proceso extenso termine. En la figura 23.27 se presenta la clase `CalculadoraPrimos`, la cual extiende a `SwingWorker` para calcular los primeros *n* números primos en un subproceso trabajador. Además de los métodos `doInBackground` y `done` que se utilizaron en el ejemplo anterior, esta clase utiliza los métodos `publish`, `process` y `setProgress` de `SwingWorker`. En este ejemplo, el método `publish` envía números primos al método `process` a medida que se van encontrando, el método `process` muestra estos números primos en un componente de la GUI, y el método `setProgress` actualiza la propiedad de progreso. Más adelante le mostraremos cómo utilizar esta propiedad para actualizar un objeto `JProgressBar`.

```java
// Fig. 23.27: CalculadoraPrimos.java
// Calcula los primeros n números primos, y muestra a medida que los va encontrando.
import javax.swing.JTextArea;
import javax.swing.JLabel;
import javax.swing.JButton;
import javax.swing.SwingWorker;
import java.util.Random;
import java.util.List;
import java.util.concurrent.ExecutionException;

public class CalculadoraPrimos extends SwingWorker< Integer, Integer >
{
   private final Random generador = new Random();
   private final JTextArea intermedioJTextArea; // muestra los números primos
   encontrados
   private final JButton obtenerPrimosJButton;
   private final JButton cancelarJButton;
   private final JLabel estadoJLabel; // muestra el estado del cálculo
   private final boolean primos[]; // arreglo booleano para buscar números primos
   private boolean detenido = false; // bandera que indica la cancelación

   // constructor
   public CalculadoraPrimos( int max, JTextArea intermedio, JLabel estado,
      JButton obtenerPrimos, JButton cancel )
   {
      intermedioJTextArea = intermedio;
```

Figura 23.27 | Calcula los primeros *n* números primos, y los muestra a medida que los va encontrando. (Parte 1 de 3).

```
      estadoJLabel = estado;
      obtenerPrimosJButton = obtenerPrimos;
      cancelarJButton = cancel;
      primos = new boolean[ max ];

      // inicializa todos los valores del arreglo primos con true
      for ( int i = 0; i < max; i ++ )
         primos[ i ] = true;
   } // fin del constructor

   // busca todos los números primos hasta max, usando la Criba de Eratóstenes
   public Integer doInBackground()
   {
      int cuenta = 0; // la cantidad de números primos encontrados

      // empezando en el tercer valor, itera a través del arreglo y pone
      // false como el valor de cualquier número mayor que sea múltiplo
      for ( int i = 2; i < primos.length; i++ )
      {
         if ( detenido ) // si se canceló un cálculo
            return cuenta;
         else
         {
            setProgress( 100 * ( i + 1 ) / primos.length );

            try
            {
               Thread.currentThread().sleep( generador.nextInt( 5 ) );
            } // fin de try
            catch ( InterruptedException ex )
            {
               estadoJLabel.setText( "Se interrumpio subproceso Trabajador" );
               return cuenta;
            } // fin de catch

            if ( primos[ i ] ) // i es primo
            {
               publish( i ); // hace a i disponible para mostrarlo en la lista de primos
               ++cuenta;

               for ( int j = i + i; j < primos.length; j += i )
                  primos[ j ] = false; // i no es primo
            } // fin de if
         } // fin de else
      } // fin de for

      return cuenta;
   } // fin del método doInBackground

   // muestra los valores publicados en la lista de números primos
   protected void process( List< Integer > valsPublicados )
   {
      for ( int i = 0; i < valsPublicados.size(); i++ )
         intermedioJTextArea.append( valsPublicados.get( i ) + "\n" );
   } // fin del método process

   // código a ejecutar cuando se completa doInBackground
   protected void done()
```

Figura 23.27 | Calcula los primeros *n* números primos, y los muestra a medida que los va encontrando. (Parte 2 de 3).

```
84      {
85         obtenerPrimosJButton.setEnabled( true ); // habilita el botón Obtener primos
86         cancelarJButton.setEnabled( false ); // deshabilita el botón Cancelar
87
88         int numPrimos;
89
90         try
91         {
92            numPrimos = get(); // obtiene el valor de retorno de doInBackground
93         } // fin de try
94         catch ( InterruptedException ex )
95         {
96            estadoJLabel.setText( "Se interrumpio mientras se esperaban los resultados." );
97            return;
98         } // fin de catch
99         catch ( ExecutionException ex )
100        {
101           estadoJLabel.setText( "Error al realizar el calculo." );
102           return;
103        } // fin de catch
104
105        estadoJLabel.setText( "Se encontraron " + numPrimos + " primos." );
106     } // fin del método done
107
108     // establece la bandera para dejar de buscar números primos
109     public void detenerCalculo()
110     {
111        detenido = true;
112     } // fin del método detenerCalculo
113  } // fin de la clase CalculadoraPrimos
```

Figura 23.27 | Calcula los primeros *n* números primos, y los muestra a medida que los va encontrando. (Parte 3 de 3).

La clase `CalculadoraPrimos` extiende a `SwingWorker` (línea 11); el primer parámetro de tipo indica el tipo de valor de retorno del método `doInBackground` y el segundo indica el tipo de los resultados intermedios que se pasan entre los métodos `publish` y `process`. En este caso, ambos parámetros de tipo son objetos `Integer`. El constructor (líneas 22 a 34) recibe como argumentos un entero que indica el límite superior de los números primos a localizar, un objeto `JTextArea` que se utiliza para mostrar los números primos en la GUI, un objeto `JButton` para iniciar un cálculo y otro para cancelarlo, y un objeto `JLabel` para mostrar el estado del cálculo.

En las líneas 32 y 33 se inicializan con `true` los elementos del arreglo `boolean` llamado `primos`. `CalculadoraPrimos` utiliza este arreglo y el algoritmo de la **Criba de Eratóstenes** (descrito en el ejercicio 7.27) para buscar todos los números primos menores que `max`. La Criba de Eratóstenes recibe una lista de números naturales de cualquier longitud y, empezando con el primer número primo, filtra todos sus múltiplos. Después avanza al siguiente número primo, que será el siguiente número que no esté filtrado todavía y elimina a todos sus múltiplos. Continúa hasta llegar al final de la lista y cuando se han filtrado todos los números que no son primos. En términos del algoritmo, empezamos con el elemento `2` del arreglo `boolean` y establecemos en `false` las celdas correspondientes a todos los valores que sean múltiplos de `2`, para indicar que pueden dividirse entre 2 y por ende, no son primos. Después avanzamos al siguiente elemento del arreglo, verificamos si es `true` y, de ser así, establecemos en `false` todos sus múltiplos para indicar que pueden dividirse entre el índice actual. Cuando se ha recorrido todo el arreglo de esta forma, todos los índices que contienen `true` son primos, ya que no tienen divisores.

En el método `doInBackground` (líneas 37 a 73), la variable de control `i` para el ciclo (líneas 43 a 70) controla el índice actual para implementar la Criba de Eratóstenes. En la línea 45 se evalúa la bandera `boolean` llamada `detenido`, la cual indica si el usuario hizo clic en el botón **Cancelar**. Si detenido es `true`, el método devuelve la cantidad de números primos encontrados hasta ese momento (línea 46) sin terminar el cálculo.

Si no se cancela el cálculo, en la línea 49 se hace una llamada al método `setProgress` para actualizar la propiedad de progreso con el porcentaje del arreglo que se ha recorrido hasta ese momento. En la línea 53 se pone en

inactividad el subproceso actual en ejecución durante un máximo de 4 milisegundos. En breve hablaremos sobre el por qué de esto. En la línea 61 se evalúa si el elemento del arreglo `primos` en el índice actual es `true` (y por ende, primo). De ser así, en la línea 63 se pasa el índice al método `publish` de manera que pueda mostrarse como resultado intermedio en la GUI, y en la línea 64 se incrementa el número de primos encontrados. En las líneas 66 y 67 se establecen en `false` todos los múltiplos del índice actual, para indicar que no son primos. Cuando se ha recorrido todo el arreglo `boolean`, el número de primos encontrados se devuelve en la línea 72.

En las líneas 76 a 80 se declara el método `process`, el cual se ejecuta en el subproceso de despachamiento de eventos y recibe su argumento `valsPublicados` del método `publish`. El paso de valores entre `publish` en el subproceso trabajador y `process` en el subproceso de despachamiento de eventos es asíncrono; `process` no se invoca necesariamente para cada una de las llamadas a `publish`. Todos los objetos `Integer` publicados desde la última llamada a `publish` se reciben como un objeto `List`, a través del método `process`. En las líneas 78 y 79 se itera a través de esta lista y se muestran los valores publicados en un objeto `JTextArea`. Como el cálculo en el método `doInBackground` progresa rápidamente, publicando valores con frecuencia, las actualizaciones al objeto `JTextArea` se pueden apilar en el subproceso de despachamiento de eventos, lo que puede provocar que la GUI tenga una respuesta lenta. De hecho, al buscar un número suficientemente largo de primos, el subproceso de despachamiento de eventos puede llegar a recibir tantas peticiones en una rápida sucesión para actualizar el objeto `JTextArea`, que el subproceso se quedará sin memoria en su cola de eventos. Ésta es la razón por la cual pusimos al subproceso trabajador en inactividad durante unos cuantos milisegundos, entre cada llamada potencial a `publish`. Se reduce la velocidad del cálculo lo suficiente como para permitir que el subproceso despachador de eventos se mantenga a la par con las peticiones para actualizar el objeto `JTextArea` con nuevos números primos, lo cual permite a la GUI actualizarse de manera uniforme y así puede permanecer con una capacidad de respuesta relativamente inmediata.

En las líneas 83 a 106 se define el método `done`. Cuando el cálculo termina o se cancela, el método `done` habilita el botón **Obtener primos** y deshabilita el botón **Cancelar** (líneas 85 y 86). En la línea 92 se obtiene el valor de retorno (el número de primos encontrados) del método `doInBackground`. En las líneas 94 a 103 se atrapan las excepciones lanzadas por el método `get` y se muestra un mensaje de error apropiado en el objeto `estadoJLabel`. Si no ocurren excepciones, en la línea 105 se establece el objeto `estadoJLabel` para indicar el número de primos encontrados.

En las líneas 109 a 112 se define el método `public detenerCalculo`, el cual se invoca cuando el usuario hace clic en el botón **Cancelar**. Este método establece la bandera `detenido` en la línea 111, de forma que `doInBackground` pueda regresar sin terminar su cálculo la próxima vez que evalúe esta bandera. Aunque `SwingWorker` cuenta con un método `cancel`, este método simplemente llama al método `interrupt` de `Thread` en el subproceso trabajador. Al utilizar la bandera `boolean` en vez del método `cancel`, podemos detener el cálculo limpiamente, devolver un valor de `doInBackground` y asegurar que se haga una llamada al método `done`, aun si el cálculo no se ejecutó hasta completarse, sin el riesgo de lanzar una excepción `InterruptedException` asociada con la acción de interrumpir el subproceso trabajador.

La clase `BuscarPrimos` (figura 23.28) muestra un objeto `JTextField` que permite al usuario escribir un número, un objeto `JButton` para empezar a buscar todos los números primos menores que ese número, y un objeto `JTextArea` para mostrar los números primos. Un objeto `JButton` permite al usuario cancelar el cálculo, y un objeto `JProgressBar` indica el progreso del cálculo. El constructor de `BuscarPrimos` (líneas 32 a 125) inicializa estos componentes y los muestra en un objeto `JFrame`, usando el esquema `BorderLayout`.

En las líneas 42 a 94 se registra el manejador de eventos para el objeto `obtenerPrimosJButton`. Cuando el usuario hace clic en este objeto `JButton`, en las líneas 47 a 49 se restablece el objeto `JProgressBar` y se borran los objetos `mostrarPrimosJTextArea` y `estadoJLabel`. En las líneas 53 a 63 se analiza el valor en el objeto `JtextField` y se muestra un mensaje de error si el valor no es un entero. En las líneas 66 a 68 se construye un nuevo objeto `CalculadoraPrimos`, el cual recibe como argumentos el entero que escribió el usuario, el objeto `mostrarPrimosJTextArea` para mostrar los números primos, el objeto `estadoJLabel` y los dos objetos `JButton`.

En las líneas 71 a 85 se registra un objeto `PropertyChangeListener` para el nuevo objeto `CalculadoraPrimos`, mediante el uso de una clase interna anónima. **`PropertyChangeListener`** es una interfaz del paquete `java.beans` que define un solo método, `propertyChange`. Cada vez que se invoca el método `setProgress` en un objeto `CalculadoraPrimos`, este objeto genera un evento `PropertyChangeEvent` para indicar que la propiedad de progreso ha cambiado. El método `propertyChange` escucha estos eventos. En la línea 78 se evalúa si un evento `PropertyChangeEvent` dado indica un cambio en la propiedad de progreso. De ser así, en la línea 80 se obtiene el nuevo valor de la propiedad y en la línea 81 se actualiza el objeto `JProgressBar` con el nuevo valor

de la propiedad de progreso. El objeto `JButton` "**Obtener primos**" se deshabilita (línea 88), de manera que sólo se pueda ejecutar un cálculo que actualice la GUI a la vez, y el objeto `JButton` "**Cancelar**" se habilita (línea 89) para permitir que el usuario detenga el cálculo antes de completarse. En la línea 91 se ejecuta el objeto `CalculadoraPrimos` para empezar a buscar números primos. Si el usuario hace clic en el objeto `cancelarJButton`, el manejador de eventos registrado en las líneas 107 a 115 llama al método `detenerCalculo` de `CalculadoraPrimos` (línea 112) y el cálculo regresa antes de terminar.

```
1  // Fig 23.28: BuscarPrimos.java
2  // Uso de un objeto SwingWorker para mostrar números primos y actualizar un objeto
3  // JProgressBar mientras se calculan los números primos.
4  import javax.swing.JFrame;
5  import javax.swing.JTextField;
6  import javax.swing.JTextArea;
7  import javax.swing.JButton;
8  import javax.swing.JProgressBar;
9  import javax.swing.JLabel;
10 import javax.swing.JPanel;
11 import javax.swing.JScrollPane;
12 import javax.swing.ScrollPaneConstants;
13 import java.awt.BorderLayout;
14 import java.awt.GridLayout;
15 import java.awt.event.ActionListener;
16 import java.awt.event.ActionEvent;
17 import java.util.concurrent.ExecutionException;
18 import java.beans.PropertyChangeListener;
19 import java.beans.PropertyChangeEvent;
20
21 public class BuscarPrimos extends JFrame
22 {
23    private final JTextField primoMayor = new JTextField();
24    private final JButton obtenerPrimosJButton = new JButton( "Obtener primos" );
25    private final JTextArea mostrarPrimosJTextArea = new JTextArea();
26    private final JButton cancelarJButton = new JButton( "Cancelar" );
27    private final JProgressBar progresoJProgressBar = new JProgressBar();
28    private final JLabel estadoJLabel = new JLabel();
29    private CalculadoraPrimos calculadora;
30
31    // constructor
32    public BuscarPrimos()
33    {
34       super( "Busqueda de primos con SwingWorker" );
35       setLayout( new BorderLayout() );
36
37       // inicializa el panel para obtener un número del usuario
38       JPanel norteJPanel = new JPanel();
39       norteJPanel.add( new JLabel( "Buscar primos menores que: " ) );
40       primoMayor.setColumns( 5 );
41       norteJPanel.add( primoMayor );
42       obtenerPrimosJButton.addActionListener(
43          new ActionListener()
44          {
45             public void actionPerformed( ActionEvent e )
46             {
47                progresoJProgressBar.setValue( 0 ); // restablece JProgressBar
48                mostrarPrimosJTextArea.setText( "" ); // borra JTextArea
49                estadoJLabel.setText( "" ); // borra JLabel
```

Figura 23.28 | Uso de un objeto `SwingWorker` para mostrar números primos y actualizar un objeto `JProgressBar` mientras se calculan los números primos. (Parte 1 de 3).

```

               int numero;

               try
               {
                  // obtiene la entrada del usuario
                  numero = Integer.parseInt(
                     primoMayor.getText() );
               } // fin de try
               catch ( NumberFormatException ex )
               {
                  estadoJLabel.setText( "Escriba un entero." );
                  return;
               } // fin de catch

               // construye un nuevo objeto CalculadoraPrimos
               calculadora = new CalculadoraPrimos( numero,
                  mostrarPrimosJTextArea, estadoJLabel, obtenerPrimosJButton,
                  cancelarJButton );

               // escucha en espera de cambios en la propiedad de la barra de progreso
               calculadora.addPropertyChangeListener(
                  new PropertyChangeListener()
                  {
                     public void propertyChange( PropertyChangeEvent e )
                     {
                        // si la propiedad modificada es progreso (progress),
                        // actualiza la barra de progreso
                        if ( e.getPropertyName().equals( "progress" ) )
                        {
                           int nuevoValor = ( Integer ) e.getNewValue();
                           progresoJProgressBar.setValue( nuevoValor );
                        } // fin de if
                     } // fin del método propertyChange
                  } // fin de la clase interna anónima
               ); // fin de la llamada a addPropertyChangeListener

               // deshabilita el botón Obtener primos y habilita el botón Cancelar
               obtenerPrimosJButton.setEnabled( false );
               cancelarJButton.setEnabled( true );

               calculadora.execute(); // ejecuta el objeto CalculadoraPrimos
            } // fin del método ActionPerformed
         } // fin de la clase interna anónima
      ); // fin de la llamada a addActionListener
      norteJPanel.add( obtenerPrimosJButton );

      // agrega un objeto JList desplazable para mostrar el resultado del cálculo
      mostrarPrimosJTextArea.setEditable( false );
      add( new JScrollPane( mostrarPrimosJTextArea,
         ScrollPaneConstants.VERTICAL_SCROLLBAR_ALWAYS,
         ScrollPaneConstants.HORIZONTAL_SCROLLBAR_NEVER ) );

      // inicializa un panel para mostrar a cancelarJButton,
      // progresoJProgressBar y estadoJLabel
      JPanel surJPanel = new JPanel( new GridLayout( 1, 3, 10, 10 ) );
      cancelarJButton.setEnabled( false );
      cancelarJButton.addActionListener(
```

Figura 23.28 | Uso de un objeto `SwingWorker` para mostrar números primos y actualizar un objeto `JProgressBar` mientras se calculan los números primos. (Parte 2 de 3).

```
            new ActionListener()
            {
               public void actionPerformed( ActionEvent e )
               {
                  calculadora.detenerCalculo(); // cancela el cálculo
               } // fin del método ActionPerformed
            } // fin de la clase interna anónima
         ); // fin de la llamada a addActionListener
         surJPanel.add( cancelarJButton );
         progresoJProgressBar.setStringPainted( true );
         surJPanel.add( progresoJProgressBar );
         surJPanel.add( estadoJLabel );

         add( norteJPanel, BorderLayout.NORTH );
         add( surJPanel, BorderLayout.SOUTH );
         setSize( 350, 300 );
         setVisible( true );
      } // fin del constructor

      // el método main empieza la ejecución del programa
      public static void main( String[] args )
      {
         BuscarPrimos aplicacion = new BuscarPrimos();
         aplicacion.setDefaultCloseOperation( EXIT_ON_CLOSE );
      } // fin de main
   } // fin de la clase BuscarPrimos
```



Figura 23.28 | Uso de un objeto `SwingWorker` para mostrar números primos y actualizar un objeto `JProgressBar` mientras se calculan los números primos. (Parte 3 de 3).

23.12 Otras clases e interfaces en `java.util.concurrent`

La interfaz `Runnable` sólo proporciona la funcionalidad más básica para la programación con subprocesamiento múltiple. De hecho, esta interfaz tiene varias limitaciones. Suponga que un objeto `Runnable` encuentra un problema y trata de lanzar una excepción verificada. El método `run` no se declara para lanzar ninguna excepción, por lo que el problema se debe manejar dentro del objeto `Runnable`; la excepción no se puede pasar al subproceso que hizo la llamada. Ahora, suponga que un objeto `Runnable` va a realizar un cálculo extenso, y que la aplicación desea obtener el resultado de ese cálculo. El método `run` no puede devolver un valor, por lo que la aplicación debe utilizar datos compartidos para pasar el valor de vuelta al subproceso que hizo la llamada. Esto también implica la sobrecarga de sincronizar el acceso a los datos. Los desarrolladores de las APIs de concurrencia que se introdujeron en Java SE 5 reconocieron estas limitaciones, y crearon una nueva interfaz para corregirlas. La interfaz **`Callable`**

(del paquete `java.util.concurrent`) declara un solo método llamado **`call`**. Esta interfaz está diseñada para que sea similar a la interfaz `Runnable` (permitir que se realice una acción de manera concurrente en un subproceso separado), pero el método `call` permite al subproceso devolver un valor o lanzar una excepción verificada.

Es probable que una aplicación que crea un objeto `Callable` necesite ejecutar el objeto `Callable` de manera concurrente con otros objetos `Runnable` y `Callable`. La interfaz `ExecutorService` proporciona el método **`submit`**, el cual ejecuta un objeto `Callable` que recibe como argumento. El método `submit` devuelve un objeto de tipo **`Future`** (del paquete `java.util.concurrent`), la cual es una interfaz que representa al objeto `Callable` en ejecución. La interfaz `Future` declara el método **`get`** para devolver el resultado del objeto `Callable` y proporciona otros métodos para administrar la ejecución de un objeto `Callable`.

23.13 Conclusión

En este capítulo aprendió que, a través de la historia, la concurrencia se ha implementado con las primitivas de sistema operativo, disponibles sólo para los programadores experimentados de sistemas, pero que Java pone la concurrencia a nuestra disposición a través del lenguaje y las APIs. También aprendió que la JVM en sí crea subprocesos para ejecutar un programa, y que también puede crear subprocesos para realizar las tareas de mantenimiento, como la recolección de basura.

Hablamos sobre el ciclo de vida de un subproceso y los estados que éste puede ocupar durante su tiempo de vida. También hablamos sobre las prioridades de los subprocesos de Java, las cuales ayudan al sistema a programar los subprocesos para su ejecución. Aprendió que debe evitar manipular las prioridades de los subprocesos en Java directamente, y aprendió también acerca de los problemas asociados con las prioridades de los subprocesos, como el aplazamiento indefinido (conocido también como inanición).

Después presentamos la interfaz `Runnable`, que se utiliza para especificar que una tarea se puede ejecutar de manera concurrente con otras tareas. El método `run` de esta interfaz se invoca mediante el subproceso que ejecuta la tarea. Mostramos cómo ejecutar un objeto `Runnable`, asociándolo con un objeto de la clase `Thread`. Después mostramos cómo usar la interfaz `Executor` para administrar la ejecución de objetos `Runnable` a través de reservas de subprocesos, las cuales pueden reutilizar los subprocesos existentes para eliminar la sobrecarga de tener que crear un nuevo subproceso para cada tarea, y pueden mejorar el rendimiento al optimizar el número de procesadores, para asegurar que el procesador se mantenga ocupado.

Aprendió que cuando varios subprocesos comparten un objeto y uno o más de ellos modifica ese objeto, pueden ocurrir resultados indeterminados, a menos que el acceso al objeto compartido se administre de manera apropiada. Le mostramos cómo resolver este problema a través de la sincronización de subprocesos, la cual coordina el acceso a los datos compartidos por varios subprocesos concurrentes. Conoció varias técnicas para realizar la sincronización: primero con la clase integrada `ArrayBlockingQueue` (la cual se encarga de todos los detalles de sincronización por usted), después con los monitores integrados de Java y la palabra clave `synchronized`, y finalmente con las interfaces `Lock` y `Condition`.

Hablamos sobre el hecho de que las GUIs de Swing no son seguras para los subprocesos, por lo que todas las interacciones con (y las modificaciones a) la GUI deben realizarse en el subproceso despachador de eventos. También vimos los problemas asociados con la realización de cálculos extensos en el subproceso despachador de eventos. Después le mostramos cómo puede usar la clase `SwingWorker` de Java SE 6 para realizar cálculos extensos en subprocesos trabajadores. Aprendió a mostrar los resultados de un objeto `SwingWorker` en una GUI cuando se completa el cálculo, y a mostrar los resultados intermedios cuando el cálculo se está realizando.

Por último, hablamos sobre las interfaces `Callable` y `Future`, las cuales nos permiten ejecutar tareas que devuelvan resultados y a obtener esos resultados, respectivamente. En el capítulo 24, Redes, utilizaremos las técnicas de subprocesamiento múltiple que presentamos aquí para que nos ayuden a crear servidores con subprocesamiento múltiple, que puedan actuar con varios clientes de manera concurrente.

Resumen

Sección 23.1 Introducción

- A través de la historia, la concurrencia se ha implementado con primitivas de sistema operativo, disponibles sólo para los programadores de sistemas experimentados.

- El lenguaje de programación Ada, desarrollado por el Departamento de defensa de los Estados Unidos, hizo que las primitivas de concurrencia estuvieran disponibles ampliamente para los contratistas de defensa dedicados a la construcción de sistemas militares de comando y control.
- Java pone las primitivas de concurrencia a disposición del programador de aplicaciones, a través del lenguaje y de las APIs. El programador especifica que una aplicación contiene subprocesos de ejecución separados, en donde cada subproceso tiene su propia pila de llamadas a métodos y su propio contador, lo cual le permite ejecutarse concurrentemente con otros subprocesos, al mismo tiempo que comparte los recursos a nivel de aplicación (como la memoria) con estos otros subprocesos.
- Además de crear subprocesos para ejecutar un programa, la JVM también puede crear subprocesos para realizar tareas de mantenimiento, como la recolección de basura.

Sección 23.2 Estados de los subprocesos: ciclo de vida de un subproceso

- En cualquier momento dado, se dice que un subproceso se encuentra en uno de varios estados de subproceso.
- Un nuevo subproceso empieza su ciclo cuando en el estado *nuevo*. Permanece en este estado hasta que el programa inicia el subproceso, con lo cual se coloca en el estado *ejecutable*. Se considera que un subproceso en el estado *ejecutable* está ejecutando su tarea.
- Algunas veces, un subproceso *ejecutable* cambia al estado *en espera* mientras espera a que otro subproceso realice una tarea. Un subproceso *en espera* regresa al estado *ejecutable* sólo cuando otro subproceso notifica al subproceso en espera que puede continuar ejecutándose.
- Un subproceso *ejecutable* puede entrar al estado *en espera* sincronizado durante un intervalo específico de tiempo. Regresa al estado *ejecutable* cuando ese intervalo de tiempo expira, o cuando ocurre el evento que está esperando.
- Un subproceso *ejecutable* puede cambiar el estado en *espera sincronizado* si proporciona un intervalo de espera opcional cuando está esperando a que otro subproceso realice una tarea. Dicho subproceso regresará al estado *ejecutable* cuando otro subproceso se lo notifique o cuando expire el intervalo sincronizado.
- Un subproceso inactivo permanece en el estado *en espera* sincronizado durante un periodo designado de tiempo, después del cual regresa al estado *ejecutable*.
- Un subproceso *ejecutable* cambia al estado *bloqueado* cuando trata de realizar una tarea que no puede completarse inmediatamente, y debe esperar temporalmente hasta que se complete esa tarea. Al estado *bloqueado* cuando trata de realizar una tarea que no puede completarse inmediatamente, y debe esperar temporalmente hasta que se complete esa tarea. En ese punto, el subproceso bloqueado cambia al estado *ejecutable*, para poder continuar su ejecución. Un subproceso *bloqueado* no puede usar un procesador, aun si hay uno disponible.
- Un subproceso *ejecutable* entra al estado *terminado* cuando completa exitosamente su tarea, o termina de alguna otra forma (tal vez debido a un error).
- A nivel del sistema operativo, el estado *ejecutable* de Java generalmente abarca dos estados separados. Cuando un subproceso cambia por primera vez al estado *ejecutable* desde el estado *nuevo*, el subproceso se encuentra en el estado listo. Un subproceso *listo* entra al estado en *ejecución* cuando el sistema operativo lo asigna a un procesador; a esto también se le conoce como despachar el subproceso.
- En la mayoría de los sistemas operativos, a cada subproceso se le otorga una pequeña cantidad de tiempo del procesador (lo cual se conoce como quantum o intervalo de tiempo) en la que debe realizar su tarea. Cuando expira su *quantum*, el subproceso regresa al estado *listo* y el sistema operativo asigna otro subproceso al procesador.
- El proceso que utiliza un sistema operativo para determinar qué subproceso debe despachar se conoce como programación de subprocesos, y depende de las prioridades de los subprocesos.

Sección 23.3 Prioridades y programación de subprocesos

- Todo subproceso en Java tiene una prioridad de subproceso (de `MIN_PRIORITY` a `MAX_PRIORITY`), la cual ayuda al sistema operativo a determinar el orden en el que se programan los subprocesos.
- De manera predeterminada, cada subproceso recibe la prioridad `NORM_PRIORITY` (una constante de 5). Cada nuevo subproceso hereda la prioridad del subproceso que lo creó.
- La mayoría de los sistemas operativos permiten que los subprocesos con igual prioridad compartan un procesador con intervalo de tiempo.
- El trabajo del programador de subprocesos de un sistema operativo es determinar cuál subproceso se debe ejecutar a continuación.
- Cuando un subproceso de mayor prioridad entra al estado *listo*, el sistema operativo generalmente sustituye el subproceso actual en *ejecución* para dar preferencia al subproceso de mayor prioridad (una operación conocida como programación preferente).

- Dependiendo del sistema operativo, los subprocesos de mayor prioridad podrían posponer (tal vez de manera indefinida) la ejecución de los subprocesos de menor prioridad. Comúnmente, a dicho aplazamiento indefinido se le conoce, en forma más colorida, como inanición.

Sección 23.4 Creación y ejecución de subprocesos

- El medio preferido de crear aplicaciones en Java con subprocesamiento múltiple es mediante la implementación de la interfaz `Runnable` (del paquete `java.lang`). Un objeto `Runnable` representa una "tarea" que puede ejecutarse concurrentemente con otras tareas.
- La interfaz `Runnable` declara el método `run`, en el cual podemos colocar el código que define la tarea a realizar. El subproceso que ejecuta un objeto `Runnable` llama al método `run` para realizar la tarea.
- Un programa no terminará sino hasta que su último subproceso termine de ejecutarse; en este punto, la JVM también terminará.
- No podemos predecir el orden en el que se van a programar los subprocesos, aun si conocemos el orden en el que se crearon y se iniciaron.
- Aunque es posible crear subprocesos en forma explícita, se recomienda utilizar la interfaz `Executor` para administrar la ejecución de objetos `Runnable` de manera automática. Por lo general, un objeto `Executor` crea y administra un grupo de subprocesos, al cual se le denomina reserva de subprocesos, para ejecutar objetos `Runnable`.
- Los objetos `Executor` pueden reutilizar los subprocesos existentes para eliminar la sobrecarga de crear un nuevo subproceso para cada tarea, y pueden mejorar el rendimiento al optimizar el número de subprocesos, con lo cual se asegura que el procesador se mantenga ocupado.
- La interfaz `Executor` declara un solo método llamado `execute`, el cual acepta un objeto `Runnable` como argumento y lo asigna a uno de los subprocesos disponibles en la reserva. Si no hay subprocesos disponibles, el objeto `Executor` crea un nuevo subproceso, o espera a que haya uno disponible.
- La interfaz `ExecutorService` (del paquete `java.util.concurrent`) extiende a la interfaz `Executor` y declara varios métodos más para administrar el ciclo de vida de un objeto `Executor`.
- Un objeto que implementa a la interfaz `ExecutorService` se puede crear mediante el uso de los métodos `static` declarados en la clase `Executors` (del paquete `java.util.concurrent`).
- El método `newCachedThreadPool` de `Executors` devuelve un objeto `ExecutorService` que crea nuevos subprocesos, según los va necesitando la aplicación.
- El método `execute` de `ExecutorService` ejecuta el objeto `Runnable` que recibe como argumento en algún momento en el futuro. El método regresa inmediatamente después de cada invocación; el programa no espera a que termine cada tarea.
- El método `shutdown` de `ExecutorService` notifica al objeto `ExecutorService` para que deje de aceptar nuevas tareas, pero continúa ejecutando las tareas que ya se hayan enviado. Una vez que se han completado todos los objetos `Runnable` enviados anteriormente, el objeto `ExecutorService` termina.

Sección 23.5 Sincronización de subprocesos

- Cuando varios subprocesos comparten un objeto, y éste puede ser modificado por uno o más de los subprocesos, pueden ocurrir resultados indeterminados a menos que el acceso al objeto compartido se administre de manera apropiada. El problema puede resolverse si se da a un subproceso a la vez el *acceso exclusivo* al código que manipula al objeto compartido. Durante ese tiempo, otros subprocesos que deseen manipular el objeto deben mantenerse en espera. Cuando el subproceso con acceso exclusivo al objeto termina de manipularlo, a uno de los subprocesos que estaba en espera se le debe permitir que continúe ejecutándose. Este proceso, conocido como sincronización de subprocesos, coordina el acceso a los datos compartidos por varios subprocesos concurrentes.
- Al sincronizar los subprocesos, podemos asegurar que cada subproceso que accede a un objeto compartido excluye a los demás subprocesos de hacerlo en forma simultánea; a esto se le conoce como exclusión mutua.
- Una manera común de realizar la sincronización es mediante los monitores integrados en Java. Cada objeto tiene un monitor y un bloqueo de monitor. El monitor asegura que el bloqueo de monitor de su objeto se mantenga por un máximo de sólo un subproceso a la vez y, por ende, se puede utilizar para imponer la exclusión mutua.
- Si una operación requiere que el subproceso en ejecución mantenga un bloqueo mientras se realiza la operación, un subproceso debe adquirir el bloqueo para poder continuar con la operación. Otros subprocesos que traten de realizar una operación que requiera el mismo *bloqueo* permanecerán bloqueados hasta que el primer subproceso libere el bloqueo, punto en el cual los subprocesos *bloqueados* pueden tratar de adquirir el bloqueo.
- Para especificar que un subproceso debe mantener un bloqueo de monitor para ejecutar un bloque de código, el código debe colocarse en una instrucción `synchronized`. Se dice que dicho código está protegido por el bloqueo de monitor; un subproceso debe adquirir el bloqueo para ejecutar las instrucciones `synchronized`.

- Las instrucciones `synchronized` se declaran mediante la palabra clave `synchronized`:

```
synchronized ( objeto )
{
    instrucciones
} // fin de la instrucción synchronized
```

  en donde *objeto* es el objeto cuyo bloqueo de monitor se va a adquirir; generalmente, *objeto* es `this` si es el objeto en el que aparece la instrucción `synchronized`.
- Un método `synchronized` es equivalente a una instrucción `synchronized` que encierra el cuerpo completo de un método, y que utiliza a `this` como el objeto cuyo bloqueo de monitor se va a adquirir.
- La interfaz `ExecutorService` proporciona el método `awaitTermination` para obligar a un programa a esperar a que los subprocesos completen su ejecución. Este método devuelve el control al que lo llamó, ya sea cuando se completen todas las tareas que se ejecutan en el objeto `ExecutorService`, o cuando se agote el tiempo de inactividad especificado. Si todas las tareas se completan antes de que se agote el tiempo de `awaitTermination`, este método devuelve `true`; en caso contrario devuelve `false`. Los dos argumentos para `awaitTermination` representan un valor de límite de tiempo y una unidad de medida especificada con una constante de la clase `TimeUnit`.
- Podemos simular la atomicidad al asegurar que sólo un subproceso lleve a cabo un conjunto de operaciones al mismo tiempo. La atomicidad se puede lograr mediante el uso de la palabra clave `synchronized` para crear una instrucción o método `synchronized`.
- Al compartir datos inmutables entre subprocesos, debemos declarar los campos de datos correspondientes como `final`, para indicar que los valores de las variables no cambiarán una vez que se inicialicen.

Sección 23.6 Relación productor/consumidor sin sincronización

- En una relación productor/consumidor con subprocesamiento múltiple, un subproceso productor genera los datos y los coloca en un objeto compartido, llamado búfer. Un subproceso consumidor lee los datos del búfer.
- Las operaciones con los datos del búfer compartidos por un subproceso productor y un subproceso consumidor son dependientes del estado; las operaciones deben proceder sólo si el búfer se encuentra en el estado correcto. Si el búfer se encuentra en un estado en el que no esté completamente lleno, el productor puede producir; si el búfer se encuentra en un estado en el que no esté completamente vacío, el consumidor puede consumir.
- Los subprocesos con acceso a un búfer deben sincronizarse para asegurar que lo datos se escriban en el búfer, o se lean del búfer sólo si éste se encuentra en el estado apropiado. Si el productor que trata de colocar los siguientes datos en el búfer determina que éste se encuentra lleno, el subproceso productor debe esperar hasta que haya espacio. Si un subproceso consumidor encuentra el búfer vacío o que los datos anteriores ya se han leído, debe también esperar hasta que haya nuevos datos disponibles.

Sección 23.7 Relación productor/consumidor: `ArrayBlockingQueue`

- Java incluye una clase de búfer completamente implementada llamada `ArrayBlockingQueue` en el paquete `java.util.concurrent`, que implementa a la interfaz `BlockingQueue`. Esta interfaz extiende a la interfaz `Queue` y declara los métodos `put` y `take`, los equivalentes con bloqueo de los métodos `offer` y `poll` de `Queue`, respectivamente.
- El método `put` coloca un elemento al final del objeto `BlockingQueue`, y espera si la cola está llena. El método `take` elimina un elemento de la parte inicial del objeto `BlockingQueue`, y espera si la cola está vacía. Estos métodos hacen que la clase `ArrayBlockingQueue` sea una buena opción para implementar un búfer compartido. Debido a que el método `put` bloquea hasta que haya espacio en el búfer para escribir datos, y el método take bloquea hasta que haya nuevos datos para leer, el productor debe producir primero un valor, el consumidor sólo consume correctamente hasta después de que el productor escribe un valor, y el productor produce correctamente el siguiente valor (después del primero) sólo hasta que el consumidor lea el valor anterior (o primero).
- `ArrayBlockingQueue` almacena los datos compartidos en un arreglo. El tamaño de este arreglo se especifica como argumento para el constructor de `ArrayBlockingQueue`. Una vez creado, un objeto `ArrayBlockingQueue` tiene su tamaño fijo y no se expandirá para dar cabida a más elementos.

Sección 23.8 Relación productor/consumidor con sincronización

- Usted puede implementar su propio búfer compartido, usando la palabra clave `synchronized` y los métodos `wait`, `notify` y `notifyAll` de `Object`, que pueden usarse con condiciones para hacer que los subprocesos esperen cuando no puedan realizar sus tareas.
- Si un subproceso obtiene el bloqueo de monitor en un objeto, y después determina que no puede continuar con su tarea en ese objeto sino hasta que se cumpla cierta condición, el subproceso puede llamar al método `wait` de

Object; esto libera el bloqueo de monitor en el objeto, y el subproceso queda en el estado *en espera* mientras el otro subproceso trata de entrar a la(s) instrucción(es) o método(s) synchronized del objeto.

- Cuando un subproceso que ejecuta una instrucción (o método) synchronized completa o cumple con la condición en la que otro subproceso puede estar esperando, puede llamar al método notify de Object para permitir que un subproceso en espera cambie al estado *ejecutable* de nuevo. En este punto, el subproceso que cambió del estado en *espera* al estado *ejecutable* puede tratar de readquirir el bloqueo de monitor en el objeto.
- Si un subproceso llama a notifyAll, entonces todos los subprocesos que esperan el bloqueo de monitor se convierten en candidatos para readquirir el bloqueo (es decir, todos cambian al estado ejecutable).

Sección 23.9 Relación productor/consumidor: búferes delimitados

- No podemos hacer suposiciones acerca de las velocidades relativas de los subprocesos concurrentes; las interacciones que ocurren con el sistema operativo, la red, el usuario y otros componentes pueden hacer que los subprocesos operen a distintas velocidades. Cuando esto ocurre, los subprocesos esperan.
- Para minimizar la cantidad de tiempo de espera para los subprocesos que comparten recursos y operan a las mismas velocidades promedio, podemos implementar un bufer delimitado. Si el productor produce temporalmente valores con más rapidez de la que el consumidor pueda consumirlos, el productor puede escribir otros valores en el espacio adicional del búfer (si hay disponible). Si el consumidor consume con más rapidez de la que el productor produce nuevos valores, el consumidor puede leer valores adicionales (si los hay) del búfer.
- La clave para usar un búfer delimitado con un productor y un consumidor que operan aproximadamente a la misma velocidad es proporcionar al búfer suficientes ubicaciones para que pueda manejar la producción "extra" anticipada.
- La manera más simple de implementar un búfer delimitado es utilizar un objeto ArrayBlockingQueue para el búfer, de manera que se haga cargo de todos los detalles de la sincronización por nosotros.

Sección 23.10 Relación productor/consumidor: las interfaces **Lock** *y* **Condition**

- Las interfaces Lock y Condition, que se introdujeron en Java SE 5, proporcionan a los programadores un control más preciso sobre la sincronización de los subprocesos, pero son más complicadas de usar.
- Cualquier objeto puede contener una referencia a un objeto que implemente a la interfaz Lock (del paquete java.util.concurrent.locks). Un subproceso llama al método lock de Lock para adquirir el bloqueo. Una vez que un subproceso obtiene un objeto Lock, este objeto no permitirá que otro subproceso obtenga el Lock sino hasta que el primer subproceso lo libere (llamando al método unlock de Lock).
- Si varios subprocesos tratan de llamar al método lock en el mismo objeto Lock y al mismo tiempo, sólo uno de estos subprocesos puede obtener el bloqueo; todos los demás se colocan en el estado *en espera* de ese bloqueo. Cuando un subproceso llama al método unlock, se libera el bloqueo sobre el objeto y un subproceso en espera que intente bloquear el objeto puede continuar.
- La clase ReentrantLock (del paquete java.util.concurrent.locks) es una implementación básica de la interfaz Lock.
- El constructor de ReentrantLock recibe un argumento boolean, el cual especifica si el bloqueo tiene una política de equidad. Si el argumento es true, la política de equidad de ReentrantLock es: "el subproceso con más tiempo de espera adquirirá el bloqueo cuando esté disponible". Dicha política de equidad garantiza que nunca ocurra el aplazamiento indefinido (también conocido como inanición). Si el argumento de la política de equidad se establece en false, no hay garantía en cuanto a cuál subproceso en espera adquirirá el bloqueo cuando esté disponible.
- Si un subproceso que posee un objeto Lock determina que no puede continuar con su tarea hasta que se cumpla cierta condición, el subproceso puede esperar en base a un objeto de condición. El uso de objetos Lock nos permite declarar de manera explícita los objetos de condición sobre los cuales un subproceso tal vez tenga que esperar.
- Los objetos de condición se asocian con un objeto Lock específico y se crean mediante una llamada al método newCondition de Lock, el cual devuelve un objeto que implementa a la interfaz Condition (del paquete java.util.concurrent.locks). Para esperar en base a un objeto de condición, el subproceso puede llamar al método await de Condition. Esto libera de inmediato el objeto Lock asociado, y coloca al subproceso en el estado *en espera*, en base a ese objeto Condition. Así, otros subprocesos pueden tratar de obtener el objeto Lock.
- Cuando un subproceso *ejecutable* completa una tarea y determina que el subproceso en espera puede ahora continuar, el subproceso *ejecutable* puede llamar al método signal de Condition para permitir que un subproceso en el estado *en espera* de ese objeto Condition regrese al estado *ejecutable*. En este punto, el subproceso que cambió del estado en espera al estado *ejecutable* puede tratar de readquirir el objeto Lock.
- Si hay varios subprocesos en un estado *en espera* de un objeto Condition cuando se hace la llamada a signal, la implementación predeterminada de Condition indica al subproceso con más tiempo de espera que debe cambiar al estado *ejecutable*.

- Si un subproceso llama al método `signalAll` de `Condition`, entonces todos los subprocesos que esperan esa condición cambian al estado *ejecutable* y se convierten en candidatos para readquirir el objeto `Lock`.
- Cuando un subproceso termina con un objeto compartido, debe llamar al método `unlock` para liberar al objeto `Lock`.
- En algunas aplicaciones, el uso de objetos `Lock` y `Condition` puede ser preferible a utilizar la palabra clave `synchronized`. Los objetos `Lock` nos permiten interrumpir a los subprocesos en espera, o especificar un tiempo límite para esperar a adquirir un bloqueo, lo cual no es posible si se utiliza la palabra clave `synchronized`. Además, un objeto `Lock` no está restringido a ser adquirido y liberado en el mismo bloque de código, lo cual es el caso con la palabra clave `synchronized`.
- Los objetos `Condition` nos permiten especificar varios objetos de condición, en base a los cuales los subprocesos pueden esperar. Por ende, es posible indicar a los subprocesos en espera que un objeto de condición específico es ahora verdadero, llamando a `signal` o `signalAll` en ese objeto `Condition`. Con la palabra clave `synchronized`, no hay forma de indicar de manera explícita la condición en la cual esperan los subprocesos y, por lo tanto, no hay forma de notificar a los subprocesos en espera de una condición específica que pueden continuar, sin también indicarlo a los subprocesos que están en espera de otras condiciones.

Sección 23.11 Subprocesamiento múltiple con GUIs

- Las aplicaciones de Swing tienen un subproceso, conocido como el subproceso de despachamiento de eventos, para manejar las interacciones con los componentes de la GUI de la aplicación. Todas las tareas que requieren interacción con la GUI de una aplicación se colocan en una cola de eventos y se ejecutan en forma secuencial, mediante el subproceso de despachamiento de eventos.
- Los componentes de GUI de Swing no son seguros para los subprocesos. La seguridad de subprocesos en aplicaciones de GUI se logra asegurando que se acceda a los componentes de Swing sólo desde el subproceso de despachamiento de eventos. A esta técnica se le conoce como confinamiento de subprocesos.
- Si una aplicación debe realizar un cálculo extenso en respuesta a una interacción con la interfaz del usuario, el subproceso de despachamiento de eventos no puede atender otras tareas en la cola de eventos, mientras se encuentre atado en ese cálculo. Esto hace que los componentes de la GUI pierdan su capacidad de respuesta. Es preferible manejar un cálculo extenso en un subproceso separado, con lo cual el subproceso de despachamiento de eventos queda libre para continuar administrando las demás interacciones con la GUI.
- Java SE 6 cuenta con la clase `SwingWorker` (en el paquete `javax.swing`), que implementa a la interfaz `Runnable`, para realizar cálculos extensos en un subproceso trabajador, y para actualizar los componentes de Swing desde el subproceso de despachamiento de eventos, con base en los resultados del cálculo.
- Para utilizar las herramientas de `SwingWorker`, cree una clase que extienda a `SwingWorker` y sobrescriba los métodos `doInBackground` y `done`. El método `doInBackground` realiza el cálculo y devuelve el resultado. El método `done` muestra los resultados en la GUI.
- `SwingWorker` es una clase genérica. Su primer parámetro de tipo indica el tipo devuelto por el método `doInBackground`; el segundo indica el tipo que se pasa entre los métodos `publish` y `process` para manejar los resultados intermedios.
- El método `doInBackground` se llama desde un subproceso trabajador. Después de que `doInBackground` regresa, el método `done` se llama desde el subproceso de despachamiento de eventos para mostrar los resultados.
- Una excepción `ExecutionException` se lanza si ocurre una excepción durante el cálculo.
- `SwingWorker` también cuenta con los métodos `publish`, `process` y `setProgress`. El método `publish` envía en forma repetida los resultados inmediatos al método `process`, el cual muestra los resultados en un componente de la GUI. El método `setProgress` actualiza la propiedad de progreso.
- El método `progress` se ejecuta en el subproceso de despachamiento de eventos y recibe datos del método `publish`. El paso de valores entre `publish` en el subproceso trabajador y `process` en el subproceso de despachamiento de eventos es asíncrono; `process` no se invoca necesariamente para cada llamada a `publish`.
- `PropertyChangeListener` es una interfaz del paquete `java.beans` que define un solo método, `propertyChange`. Cada vez que se invoca el método `setProgress`, se genera un evento `PropertyChangedEvent` para indicar que la propiedad de progreso ha cambiado.

Sección 23.12 Otras clases e interfaces en `java.util.concurrent`

- La interfaz `Callable` (del paquete `java.util.concurrent`) declara un solo método llamado `call`. Esta interfaz está diseñada para que sea similar a la interfaz `Runnable` (permitir que se realice una acción de manera concurrente en un subproceso separado), pero el método `call` permite al subproceso devolver un valor o lanzar una excepción verificada.

- Es probable que una aplicación que crea un objeto `Callable` necesite ejecutar el objeto `Callable` de manera concurrente con otros objetos `Runnable` y `Callable`. La interfaz `ExecutorService` proporciona el método `submit`, el cual ejecuta un objeto `Callable` que recibe como argumento.
- El método `submit` devuelve un objeto de tipo `Future` (del paquete `java.util.concurrent`), que representa al objeto `Callable` en ejecución. La interfaz `Future` declara el método `get` para devolver el resultado del objeto `Callable` y proporciona otros métodos para administrar la ejecución de un objeto `Callable`.

Terminología

adquirir el bloqueo
aplazamiento indefinido
`ArrayBlockingQueue`, clase
`await`, método de la interfaz `Condition`
`awaitTermination`, método de la interfaz `ExecutorService`
`BlockingQueue`, interfaz
bloqueado, estado
bloqueo de monitor
bloqueo intrínseco
búfer
búfer circular
búfer delimitado
`call`, método de la interfaz `Callable`
`Callable`, interfaz
cola de prioridad multinivel
concurrencia
`Condition`, interfaz
confinamiento de subprocesos
consumidor
datos mutables
dependencia de estados
despachamiento de un subproceso
ejecutable, estado
en ejecución, estado
en espera sincronizado, estado
en espera, estado
estado de un subproceso
exclusión mutua
`execute`, método de la interfaz `Executor`
`Executor`, interfaz
`Executors`, clase
`ExecutorService`, interfaz
`Future`, interfaz
`get`, método de la interfaz `Future`
`IllegalMonitorStateException`, clase
inanición
interbloqueo
`interrupt`, método de la clase `Thread`
`InterruptedException`, clase
intervalo de inactividad
intervalo de tiempo
`java.util.concurrent`, paquete
`java.util.concurrent.locks`, paquete
listo, estado
`Lock`, interfaz
`lock`, método de la interfaz `Lock`
monitor
`newCachedThreadPool`, método de la clase `Executors`
`newCondition`, método de la interfaz `Lock`
`notify`, método de la clase `Object`
`notifyAll`, método de la clase `Object`
nuevo, estado
objeto de condición
operación atómica
operaciones en paralelo
política de equidad de un bloqueo
prioridad de subprocesos
productor
programación cíclica (round-robin)
programación concurrente
programación de subprocesos
programación preferente
programador de subprocesos
`propertyChange`, método de la interfaz `PropertyChangeListener`
`PropertyChangeListener`, interfaz
protegido por un bloqueo
`put`, método de la interfaz `BlockingQueue`
quantum
recolección de basura
`ReentrantLock`, clase
relación productor/consumidor
reserva de subprocesos
`run`, método de la interfaz `Runnable`
`Runnable`, interfaz
seguro para los subprocesos
`shutdown`, método de la clase `ExecutorService`
`signal`, método de la clase `Condition`
`signalAll`, método de la clase `Condition`
sincronización
sincronización de subprocesos
`size`, método de la clase `ArrayBlockingQueue`
`sleep`, método de la clase `Thread`
`submit`, método de la clase `ExecutorService`
subprocesamiento múltiple
subproceso
subproceso consumidor
subproceso de despachamiento de eventos
subproceso inactivo
subproceso principal
subproceso productor
`SwingWorker`, clase
`synchronized`, instrucción
`synchronized`, método
`synchronized`, palabra clave

`take`, método de la interfaz `BlockingQueue`
terminado, estado
`Thread`, clase
`unlock`, método de la interfaz `Lock`
valor pasado
`wait`, método de la clase `Object`

Ejercicios de autoevaluación

23.1 Complete las siguientes oraciones:

a) C y C++ son lenguajes con subprocesamiento ________________, mientras que Java es un lenguaje con subprocesamiento ________________.
b) Un subproceso entra al estado *terminado* cuando ________________.
c) Para detenerse durante cierto número designado de milisegundos y reanudar su ejecución, un subproceso debe llamar al método ________________ de la clase ________________.
d) El método ________________ de la clase `Condition` pasa a un solo subproceso en el estado *en espera* de un objeto, al estado *ejecutable*.
e) El método ________________ de la clase `Condition` pasa a todos los subprocesos en el estado *en espera* de un objeto, al estado *ejecutable*.
f) Un subproceso ________________ entra al estado ________________ cuando completa su tarea, o cuando termina de alguna otra forma.
g) Un subproceso *ejecutable* puede entrar al estado ________________ durante un intervalo específico.
h) A nivel del sistema operativo, el estado *ejecutable* en realidad abarca dos estados separados, ________________ y ________________.
i) Los objetos `Runnable` se ejecutan usando una clase que implementa a la interfaz ________________.
j) El método ________________ de `ExecutorService` termina cada subproceso en un objeto `ExecutorService` tan pronto como termina de ejecutar su objeto `Runnable` actual, si lo hay.
k) Un subproceso puede llamar al método ________________ en un objeto `Condition` para liberar el objeto `Lock` asociado, y colocar ese subproceso en el estado ________________.
l) En una relación ____________, la porción correspondiente al ________________ de una aplicación genera datos y los almacena en un objeto compartido, y la porción correspondiente al ________________ de una aplicación lee datos del objeto compartido.
m) La clase ________________ implementa a la interfaz `BlockingQueue`, usando un arreglo.
n) La palabra clave ________________ indica que sólo se debe ejecutar un subproceso a la vez en un objeto.

23.2 Conteste con *verdadero* o *falso* a cada una de las siguientes proposiciones; en caso de ser *falso*, explique por qué.

a) Un subproceso no es *ejecutable* si ha terminado.
b) Un subproceso *ejecutable* de mayor prioridad tiene preferencia sobre los subprocesos de menor prioridad.
c) Algunos sistemas operativos utilizan el intervalo de tiempo con subprocesos. Por lo tanto, pueden permitir que los subprocesos desplacen a otros subprocesos de la misma prioridad.
d) Cuando expira el quantum de un subproceso, éste regresa al estado *ejecutable*, a medida que el sistema operativo asigna el subproceso a un procesador.
e) En un sistema con un solo procesador sin intervalo de tiempo, cada subproceso en un conjunto de subprocesos de igual prioridad (sin otros subprocesos presentes) se ejecuta hasta terminar, antes de que otros subprocesos de igual prioridad tengan oportunidad de ejecutarse.

Respuestas a los ejercicios de autoevaluación

23.1 a) simple, múltiple. b) termina su método `run`. c) `sleep`, `Thread`. d) `signal`. e) `signalAll`, f) *ejecutable*, *terminado*. g) *en espera sincronizado*. h) *listo*, *en ejecución*. i) `Executor`. j) `shutdown`. k) `await`, *en espera*. l) productor/consumidor, productor, consumidor. m) `ArrayBlockingQueue`. n) `synchronized`.

23.2 a) Verdadero. b) Verdadero. c) Falso. El intervalo de tiempo permite a un subproceso ejecutarse hasta que expira su porción de tiempo (o quantum). Después pueden ejecutarse otros subprocesos de igual prioridad. d) Falso. Cuando expira el quantum de un subproceso, éste regresa al estado *listo* y el sistema operativo asigna otro subproceso al procesador. e) Verdadero.

Ejercicios

23.3 Conteste con *verdadero* o *falso* a cada una de las siguientes proposiciones; en caso de ser *falso*, explique por qué.

a) El método `sleep` no consume tiempo del procesador cuando un subproceso está inactivo.
b) Al declarar un método como `synchronized` se garantiza que no pueda ocurrir el interbloqueo.
c) Una vez que un subproceso obtiene un objeto `Lock`, éste no permitirá que otro subproceso obtenga el bloqueo sino hasta que el primer subproceso lo libere.
d) Los componentes de Swing son seguros para los subprocesos.

23.4 Defina cada uno de los siguientes términos.

a) subproceso
b) subprocesamiento múltiple
c) estado *ejecutable*
d) estado *en espera* sincronizado
e) programación preferente
f) interfaz `Runnable`
g) método `notifyAll`
h) relación productor/consumidor
i) quantum

23.5 Describa cada uno de los siguientes términos en el contexto de los mecanismos de subprocesamiento en Java:

a) `synchronized`
b) productor
c) consumidor
d) `wait`
e) `notify`
f) `Lock`
g) `Condition`

23.6 Enliste las razones para entrar al estado *bloqueado*. Para cada una de éstas, describa la forma en que el programa comúnmente sale del estado *bloqueado* y entra al estado *ejecutable*.

23.7 Dos problemas que pueden ocurrir en sistemas que permiten a los subprocesos esperar son: el interbloqueo, en el cual uno o más subprocesos esperarán para siempre un evento que no puede ocurrir, y el aplazamiento indefinido, en donde uno o más subprocesos se retrasarán durante cierto tiempo, sin saber cuánto. Dé un ejemplo de cómo cada uno de estos problemas puede ocurrir en los programas de Java con subprocesamiento múltiple.

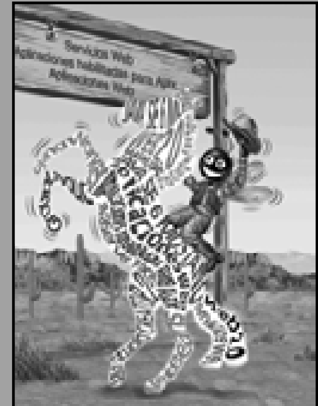
23.8 Escriba un programa para rebotar una pelota azul dentro de un objeto `JPanel`. La pelota deberá empezar a moverse con un evento `mousePressed`. Cuando la pelota pegue en el borde del objeto `JPanel`, deberá rebotar y continuar en la dirección opuesta. La pelota debe actualizarse mediante el uso de un objeto `Runnable`.

23.9 Modifique el programa del ejercicio 23.8 para agregar una nueva pelota cada vez que el usuario haga clic con el ratón. Proporcione un mínimo de 20 pelotas. Seleccione al azar el color para cada nueva pelota.

23.10 Modifique el programa del ejercicio 23.9 para agregar sombras. A medida que se mueva una pelota, dibuje un óvalo relleno de color negro en la parte inferior del objeto `JPanel`. Tal vez sería conveniente agregar un efecto tridimensional, incrementando o decrementando el tamaño de cada pelota cuando ésta pegue en el borde del subprograma.

24

Redes



Si la presencia de electricidad puede hacerse visible en cualquier parte de un circuito, no veo la razón por la cual la inteligencia no pueda transmitirse instantáneamente mediante la electricidad.

—Samuel F. B. Morse

El protocolo es todo.

—Francois Giuliani

Lo que las redes de vías de ferrocarril, carreteras y canales fueron en otra época, las redes de telecomunicaciones, información y computación lo son hoy.

—Bruno Kreisky

El puerto está cerca, escucho las campanas, toda la gente está exultando.

—Walt Whitman

OBJETIVOS

En este capítulo aprenderá a:

- Comprender el trabajo en red en Java con URLs, sockets y datagramas.
- Implementar aplicaciones de red en Java, mediante el uso de sockets y datagramas.
- Implementar clientes y servidores en Java, que se comuniquen entre sí.
- Saber cómo implementar aplicaciones de colaboración basada en red.
- Construir un servidor con subprocesamiento múltiple.

Plan general

24.1 Introducción

Hay mucha emoción en cuanto a Internet y World Wide Web. Internet enlaza la "información de todo el mundo". World Wide Web facilita el uso de Internet y le proporciona los beneficios de multimedia. Las organizaciones ven a Internet y a la Web como algo crucial para sus estrategias de sistemas de información. Java proporciona una variedad de herramientas de red integradas, las cuales facilitan el desarrollo de aplicaciones basadas en Internet y Web. Java puede permitir a los programas buscar información en todo el mundo y colaborar con programas que se ejecutan en otras computadoras a nivel internacional, nacional o solamente dentro de una organización. Java puede permitir que los applets (subprogramas) y aplicaciones se comuniquen entre sí (lo cual está sujeto a ciertas restricciones de seguridad).

Las redes son un tema masivo y complejo. Los estudiantes de ciencias computacionales e ingeniería computacional, por lo general, toman un curso de nivel superior (con duración de un semestre completo) sobre redes de computadoras y continúan estudiando a nivel de graduados. Java se utiliza comúnmente como un vehículo de implementación en cursos de redes computacionales. En este libro le presentamos una porción de los conceptos y herramientas de Java para trabajo en red.

En Java, las herramientas de red fundamentales se declaran mediante clases e interfaces del paquete `java.net`, mediante el cual Java ofrece **comunicaciones basadas en flujos** que permiten a las aplicaciones ver las redes como flujos de datos. Las clases e interfaces del paquete `java.net` también ofrecen **comunicaciones basadas en paquetes**, para transmitir **paquetes** individuales de información; esto se utiliza comúnmente para transmitir audio y video a través de Internet. En este capítulo mostraremos cómo crear y manipular sockets, y cómo comunicarnos con los paquetes de datos.

Nuestra discusión sobre las redes se enfoca en ambos lados de una **relación cliente-servidor**. El **cliente** solicita que se realice cierta acción, y el **servidor** realiza la acción y responde al cliente. Una implementación común del modelo de petición-respuesta se da entre los navegadores y servidores Web. Cuando un usuario selecciona un sitio Web para navegar mediante un navegador (la aplicación cliente), se envía una petición al servidor Web apropiado (la aplicación servidor). Por lo general, el servidor responde al cliente enviando una página Web de HTML apropiada.

Presentaremos las **comunicaciones basadas en sockets** en Java, las cuales permiten a las aplicaciones ver las redes como si fueran E/S de archivo; un programa puede leer de un **socket** o escribir en un socket de una manera tan simple como leer o escribir en un archivo. El socket es simplemente una construcción de software que representa un extremo de una conexión. Mostraremos cómo crear y manipular sockets de flujo y sockets de datagrama.

Con los **sockets de flujo**, un proceso establece una **conexión** con otro proceso. Mientras la conexión está en pie, los datos fluyen entre los procesos en **flujos** continuos. Se dice que los sockets de flujo proporcionan un **servicio orientado a la conexión**. El protocolo utilizado para la transmisión es el popular **TCP (Protocolo de control de transmisión)**.

Con los **sockets de datagrama** se transmiten **paquetes** individuales de información. Esto no es adecuado para los programadores cotidianos, ya que el protocolo utilizado **(UDP, el Protocolo de datagramas de usuario)**

es un **servicio sin conexión** y no garantiza que los paquetes lleguen en un orden específico. Con el UDP pueden perderse paquetes, o incluso duplicarse. Se requiere de programación adicional por parte del programador para tratar con estos problemas (si el programador así lo quiere). UDP es más apropiado para aplicaciones de red en las que no se requiere la comprobación de errores y la confiabilidad de TCP. Los sockets de flujo y el protocolo TCP son más convenientes para la gran mayoría de programadores de Java.



Tip de rendimiento 24.1

Los servicios sin conexión generalmente ofrecen un mayor rendimiento, pero son menos confiables que los servicios orientados a las conexiones.



Tip de portabilidad 24.1

TCP, UDP y los protocolos relacionados permiten que una gran variedad de sistemas computacionales heterogéneos (es decir, sistemas computacionales con diferentes procesadores y sistemas operativos) se comuniquen unos con otros.

También presentaremos un ejemplo práctico en el que implementamos una aplicación cliente/servidor para conversar (chat), similar a los populares servicios de mensajería instantánea en Web hoy en día. Este ejemplo práctico se proporciona como bono Web en `www.deitel.com/books/jhtp7/`. La aplicación incorpora muchas técnicas de red que introduciremos en este capítulo. El programa también introduce el término **transmisión múltiple (multicasting)**, en donde un servidor puede publicar información y los clientes pueden suscribirse a esa información. Cada vez que el servidor publica más información, todos los suscriptores la reciben. A lo largo de los ejemplos de este capítulo veremos que muchos de los detalles del trabajo en red se manejan mediante las APIs de Java.

24.2 Manipulación de URLs

Internet ofrece muchos protocolos. El **Protocolo de transferencia de hipertexto (HTTP)** que forma la base de World Wide Web, utiliza **URIs (Identificadores uniformes de recursos)** para identificar datos en Internet. Los URIs que especifican las ubicaciones de documentos se conocen como **URLs (Localizadores uniformes de recursos)**. Los URLs comunes hacen referencia a archivos o directorios y pueden hacer referencia a objetos que realizan tareas complejas, como búsquedas en bases de datos y en Internet. Si usted conoce el URL de HTTP de un documento HTML que esté públicamente disponible en Web, puede acceder a esos datos a través de HTTP.

Java facilita la manipulación de URLs. Si se utiliza un URL que haga referencia a la ubicación exacta de un recurso (como una página Web) como un argumento para el método **`showDocument`** de la interfaz **`AppletContext`**, el navegador en el que se ejecuta el applet mostrará ese recurso. El applet de las figuras 24.1 y 24.2 demuestra el uso de las herramientas simples de red. El applet permite al usuario seleccionar una página Web de un objeto `JList`, y hace que el navegador muestre la página correspondiente. En este ejemplo, el trabajo en red es realizado por el navegador.

Este applet aprovecha los **parámetros de applet** especificados en el documento HTML que invoca al applet. Al navegar por World Wide Web, a menudo se encontrará applets en el dominio público; puede utilizarlos sin costo en sus propias páginas Web (generalmente a cambio de dar créditos al creador del applet). Muchos applets pueden personalizarse mediante los parámetros que se proporcionan desde el archivo HTML que invoca al applet. Por ejemplo, en la figura 24.1 se muestra el HTML que invoca al objeto `SelectorSitios` en la figura 24.2.

```
<html>
<title>Selector de sitios</title>
<body>
   <applet code = "SelectorSitios.class" width = "300" height = "75">
      <param name = "titulo0" value = "Página inicial de Java">
      <param name = "ubicacion0" value = "http://java.sun.com/">
      <param name = "titulo1" value = "Deitel">
```

Figura 24.1 | Documento HTML para cargar el applet `SelectorSitios`. (Parte 1 de 2).

```
      <param name = "ubicacion1" value = "http://www.deitel.com/">
      <param name = "titulo2" value = "JGuru">
      <param name = "ubicacion2" value = "http://www.jGuru.com/">
      <param name = "titulo3" value = "JavaWorld">
      <param name = "ubicacion3" value = "http://www.javaworld.com/">
   </applet>
</body>
</html>
```

Figura 24.1 | Documento HTML para cargar el applet `SelectorSitios`. (Parte 2 de 2).

```
// Fig. 24.2: SelectorSitios.java
// Este programa carga un documento de un URL.
import java.net.MalformedURLException;
import java.net.URL;
import java.util.HashMap;
import java.util.ArrayList;
import java.awt.BorderLayout;
import java.applet.AppletContext;
import javax.swing.JApplet;
import javax.swing.JLabel;
import javax.swing.JList;
import javax.swing.JScrollPane;
import javax.swing.event.ListSelectionEvent;
import javax.swing.event.ListSelectionListener;

public class SelectorSitios extends JApplet
{
   private HashMap< Object, URL > sitios; // nombres de sitios y URLs
   private ArrayList< String > nombresSitios; // nombres de sitios
   private JList selectorSitios; // lista de sitios a elegir

   // lee los parámetros de HTML y establece la GUI
   public void init()
   {
      sitios = new HashMap< Object, URL >(); // crea objeto HashMap
      nombresSitios = new ArrayList< String >(); // crea objeto ArrayList

      // obtiene los parámetros del documento de HTML
      obtenerSitiosDeParametrosHTML();

      // crea componentes de GUI e interfaz de esquema
      add( new JLabel( "Seleccione un sitio para navegar" ), BorderLayout.NORTH );

      selectorSitios = new JList( nombresSitios.toArray() ); // llena el objeto JList
      selectorSitios.addListSelectionListener(
         new ListSelectionListener() // clase interna anónima
         {
            // va al sitio seleccionado por el usuario
            public void valueChanged( ListSelectionEvent evento )
            {
               // obtiene el nombre del sitio seleccionado
               Object objeto = selectorSitios.getSelectedValue();

               // usa el nombre del sitio para localizar el URL correspondiente
               URL nuevoDocumento = sitios.get( objeto );

```

Figura 24.2 | Cómo cargar un documento de un URL a un navegador. (Parte 1 de 3).

```
               // obtiene el contenedor de applets
               AppletContext navegador = getAppletContext();

               // indica al contenedor de applets que cambie de página
               navegador.showDocument( nuevoDocumento );
            } // fin del método valueChanged
         } // fin de la clase interna anónima
      ); // fin de la llamada a addListSelectionListener

      add( new JScrollPane( selectorSitios ), BorderLayout.CENTER );
   } // fin del método init

   // obtiene los parámetros del documento de HTML
   private void obtenerSitiosDeParametrosHTML()
   {
      String titulo; // titulo del sitio
      String ubicacion; // ubicacion del sitio
      URL url; // URL de la ubicación
      int contador = 0; // cuenta el número de sitios

      titulo = getParameter( "titulo" + contador ); // obtiene el título del primer sitio

      // itera hasta que no haya más parámetros en el documento de HTML
      while ( titulo != null )
      {
         // obtiene la ubicación del sitio
         ubicacion = getParameter( "ubicacion" + contador );

         try // coloca titulo/URL en objeto HashMap y titulo en objeto ArrayList
         {
            url = new URL( ubicacion ); // convierte la ubicación en URL
            sitios.put( titulo, url ); // coloca titulo/URL en objeto HashMap
            nombresSitios.add( titulo ); // coloca titulo en objeto ArrayList
         } // fin de try
         catch ( MalformedURLException excepcionURL )
         {
            excepcionURL.printStackTrace();
         } // fin de catch

         contador++;
         titulo = getParameter( "titulo" + contador ); // obtiene el título del
         siguiente sitio
      } // fin de while
   } // fin del método obtenerSitiosDeParametrosHTML
} // fin de la clase SelectorSitios
```

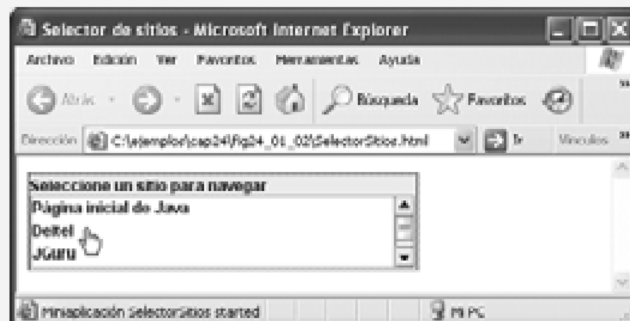


Figura 24.2 | Cómo cargar un documento de un URL a un navegador. (Parte 2 de 3).



Figura 24.2 | Cómo cargar un documento de un URL a un navegador. (Parte 3 de 3).

El documento HTML contiene ocho parámetros especificados con la **marca param**; estas líneas deben aparecer entre las marcas **applet** inicial y final. El applet puede leer estos valores y utilizarlos para personalizarse a sí mismo. Puede aparecer cualquier número de marcas param entre las marcas applet inicial y final. Cada parámetro tiene un **nombre** y un **valor**. El método **getParameter** de Applet obtiene el valor (value) asociado con un nombre de parámetro específico y lo devuelve como una cadena. El argumento que se pasa a getParameter es una cadena que contiene el nombre del parámetro en el elemento param. En este ejemplo, los parámetros representan el título y la ubicación de cada sitio Web que puede seleccionar el usuario. Los parámetros especificados para este applet se nombran com titulo#, en donde el valor de # empieza en 0 y se incrementa en uno para cada nuevo título. Cada título debe tener un parámetro de ubicación correspondiente, de la forma ubicacion#, en donde el valor de # empieza en 0 y se incrementa en uno para cada nueva ubicación. La instrucción

```
String titulo = getParameter( "titulo0" );
```

obtiene el valor asociado con el parámetro "titulo0" y lo asigna a la referencia titulo. Si no hay una marca param que contenga el parámetro especificado, getParameter devuelve null.

El applet (figura 24.2) obtiene del documento HTML (figura 24.1) las opciones a mostrar en el objeto JList del applet. La clase SelectorSitios utiliza un objeto HashMap (paquete java.util) para almacenar los nombres de sitios y sus URLs. En este ejemplo, la *clave* es la cadena en el objeto JList que representa el nombre del sitio Web, y el *valor* es un objeto URL que almacena la ubicación del sitio Web que se mostrará en el navegador.

La clase SelectorSitios también contiene un objeto ArrayList (paquete java.util) en donde se colocan los nombres de los sitios, de manera que se puedan utilizar para inicializar el objeto JList (una versión del constructor de JList recibe un arreglo de objetos Object, el cual es devuelto por el método toArray de ArrayList. Un objeto ArrayList es un arreglo de referencias que puede cambiar su tamaño en forma dinámica. La clase ArrayList proporciona el método add para agregar un nuevo elemento al final del objeto ArrayList. (En el capítulo 19 hablamos sobre las clases ArrayList y HashMap).

En las líneas 25 a 26 del método init del applet (líneas 23 a 57) se crea un objeto HashMap y un objeto ArrayList. En la línea 29 se hace una llamada a nuestro método utilitario obtenerSitiosDeParametrosHTML (declarado en las líneas 60 a 89) para obtener los parámetros de HTML del documento HTML que invocó al applet.

En el método obtenerSitiosDeParametrosHTML se utiliza el método getParameter de Applet (línea 67) para obtener el título de un sitio Web. Si el titulo no es null, el ciclo de las líneas 70 a 88 empieza a ejecutarse. En la línea 73 se utiliza el método getParameter de Applet para obtener la ubicación del sitio Web. En la línea 77 se utiliza la ubicacion como el valor de un nuevo objeto URL. El constructor de URL determina

si su argumento representa a un URL válido. Si no es así, el constructor de URL lanza una excepción **MalformedURLException**. Observe que el constructor de URL debe ser llamado en un bloque **try**. Si el constructor de URL genera una excepción MalformedURLException, la llamada a printStackTrace (línea 83) hace que el programa imprima un rastreo de la pila en la consola de Java. En equipos Windows, para ver la consola de Java hay que hacer clic con el botón derecho del ratón en el icono de Java que se encuentra en el área de notificación de la barra de tareas. Después el programa trata de obtener el título del siguiente sitio Web. El programa no agrega el sitio del URL inválido al objeto HashMap, por lo que ese título no se mostrará en el objeto JList.

Para un URL correcto, en la línea 78 se colocan el titulo y el URL en el objeto HashMap, y en la línea 79 se agrega el titulo al objeto ArrayList. En la línea 87 se obtiene el siguiente título del documento HTML. Cuando la llamada a getParameter en la línea 87 devuelve null, el ciclo termina.

Cuando el método obtenerSitiosDeParametrosHTML regresa a init, en las líneas 32 a 56 se construye la GUI del applet. En la línea 32 se agrega la etiqueta JLabel "Seleccione un sitio para navegar" a la sección NORTH del esquema BorderLayout del objeto JFrame. En la línea 34 se crea un objeto JList llamado selectorSitios para permitir al usuario seleccionar una página Web y verla. En las líneas 35 a 54 se registra un objeto ListSelectionListener para manejar los eventos de selectorSitios. En la línea 56 se agrega selectorSitios a la sección CENTER del esquema BorderLayout del objeto JFrame.

Cuando el usuario selecciona uno de los sitios Web listados en selectorSitios, el programa llama al método valueChanged (líneas 39 a 52). En la línea 42 se obtiene del objeto JList el nombre del sitio seleccionado. En la línea 45 se pasa el nombre del sitio seleccionado (la *clave*) al método get de HashMap, el cual localiza y devuelve una referencia al objeto URL correspondiente (el *valor*) que se asigna a la referencia nuevoDocumento.

En la línea 48 se utiliza el método **getAppletContext** de Applet para obtener una referencia a un objeto AppletContext que representa el contenedor del applet. En la línea 51 se utiliza la referencia navegador de AppletContext para invocar el método **showDocument**, el cual recibe un objeto URL como argumento, y lo pasa al objeto AppletContext (es decir, el navegador). El navegador muestra en su ventana actual el recurso Web asociado con ese URL. En este ejemplo, todos los recursos son documentos HTML.

Para los programadores familiarizados con los **marcos de HTML**, hay una segunda versión del método showDocument de AppletContext que permite a un applet especificar lo que se conoce como el **marco de destino**, en el cual se mostrará el recurso Web. Esta segunda versión recibe dos argumentos: un objeto URL que especifica el recurso a mostrar, y una cadena que representa el marco de destino. Hay algunos marcos de destino especiales que pueden utilizarse como el segundo argumento. El marco de destino **_blank** ocasiona que se muestre el contenido del URL especificado en una nueva ventana del navegador Web. El marco de destino **_self** especifica que el contenido del URL especificado debe mostrarse en el mismo marco que el applet (la página HTML del applet se reemplaza en este caso). El marco de destino **_top** especifica que el navegador debe eliminar los marcos actuales en la ventana del navegador y después mostrar el contenido del URL especificado en la ventana actual. [*Nota:* si le interesa aprender más acerca de HTML, el CD que se incluye con este libro contiene tres capítulos de nuestro libro *Internet and World Wide Web How to Program, Tercera edición*, que introducen la versión actual de HTML (conocida como XHTML) y la herramienta para formato de páginas Web conocida como Hojas de estilo en cascada (CSS)].



Tip para prevenir errores 24.1

El applet de la figura 24.2 debe ejecutarse desde un navegador Web como Mozilla o Microsoft Internet Explorer, para que se pueda ver el resultado de mostrar otra página Web. El appletviewer *es capaz de ejecutar applets solamente; ignora todas las demás marcas de HTML. Si los sitos Web en el programa incluyeran applets de Java, sólo aparecerían esos applets en el* appletviewer *cuando el usuario seleccionara un sitio Web. Cada applet se ejecutaría en una ventana separada del* appletviewer.

24.3 Cómo leer un archivo en un servidor Web

En nuestro siguiente ejemplo, una vez más ocultamos los detalles relacionados con la red. La aplicación de la figura 24.3 utiliza el componente de la GUI de Swing **JEditorPane** (del paquete javax.swing) para mostrar el contenido de un archivo en un servidor Web. El usuario introduce un URL en el objeto JTextField que se encuentra en la parte superior de la ventana, y la aplicación muestra el documento correspondiente (si es que existe) en el objeto JEditorPane. La clase JEditorPane puede desplegar tanto texto simple como texto con formato HTML (como se muestra en las dos capturas de pantalla de la figura 24.4), por lo que esta aplicación actúa como

un navegador Web simple. La aplicación también demuestra cómo procesar eventos **`HyperlinkEvent`** cuando el usuario hace clic en un hipervínculo en el documento HTML. Las técnicas que se muestran en este ejemplo también pueden usarse en applets. Sin embargo, a los applets sólo se les permite leer archivos en el servidor del cual se hayan descargado.

La clase de aplicación `LeerArchivoServidor` contiene el objeto `JTextField` llamado `campoIntroducir`, en el cual el usuario escribe el URL del archivo a leer, y el objeto `JEditorPane` llamado `areaContenido` para mostrar el contenido del archivo. Cuando el usuario oprime la tecla *Intro* en `campoIntroducir`, el programa

```java
// Fig. 24.3: LeerArchivoServidor.java
// Uso de un objeto JEditorPane para mostrar el contenido de un archivo en un servidor Web.
import java.awt.BorderLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.io.IOException;
import javax.swing.JEditorPane;
import javax.swing.JFrame;
import javax.swing.JOptionPane;
import javax.swing.JScrollPane;
import javax.swing.JTextField;
import javax.swing.event.HyperlinkEvent;
import javax.swing.event.HyperlinkListener;

public class LeerArchivoServidor extends JFrame
{
   private JTextField campoIntroducir; // objeto JTextField para escribir el nombre del
   sitio
   private JEditorPane areaContenido; // para mostrar un sitio Web

   // establece la GUI
   public LeerArchivoServidor()
   {
      super( "Navegador Web simple" );

      // crea campoIntroducir y registra su componente de escucha
      campoIntroducir = new JTextField( "Escriba el URL del archivo" );
      campoIntroducir.addActionListener(
         new ActionListener()
         {
            // obtiene el documento especificado por el usuario
            public void actionPerformed( ActionEvent evento )
            {
               obtenerLaPagina( evento.getActionCommand() );
            } // fin del método actionPerformed
         } // fin de la clase interna
      ); // fin de la llamada a addActionListener

      add( campoIntroducir, BorderLayout.NORTH );

      areaContenido = new JEditorPane(); // crea areaContenido
      areaContenido.setEditable( false );
      areaContenido.addHyperlinkListener(
         new HyperlinkListener()
         {
            // si el usuario hizo clic en un hipervínculo, va a la página especificada
            public void hyperlinkUpdate( HyperlinkEvent evento )
            {
               if ( evento.getEventType() ==
```

Figura 24.3 | Cómo leer un archivo, abriendo una conexión a través de un URL. (Parte 1 de 2).

```
                  HyperlinkEvent.EventType.ACTIVATED )
                 obtenerLaPagina( evento.getURL().toString() );
           } // fin del método hyperlinkUpdate
        } // fin de la clase interna anónima
      ); // fin de la llamada a addHyperlinkListener

      add( new JScrollPane( areaContenido ), BorderLayout.CENTER );
      setSize( 400, 300 ); // establece el tamaño de la ventana
      setVisible( true ); // muestra la ventana
   } // fin del constructor de LeerArchivoServidor

   // carga el documento
   private void obtenerLaPagina( String ubicacion )
   {
      try // carga el documento y muestra la ubicación
      {
         areaContenido.setPage( ubicacion ); // establece la página
         campoIntroducir.setText( ubicacion ); // establece el texto
      } // fin de try
      catch ( IOException excepcionES )
      {
         JOptionPane.showMessageDialog( this,
            "Error al obtener el URL especificado", "URL incorrecto",
            JOptionPane.ERROR_MESSAGE );
      } // fin de catch
   } // fin del método obtenerLaPagina
} // fin de la clase LeerArchivoServidor
```

Figura 24.3 | Cómo leer un archivo, abriendo una conexión a través de un URL. (Parte 2 de 2).

```
// Fig. 24.4: PruebaLeerArchivoServidor.java
// Crea e inicia un objeto LeerArchivoServidor.
import javax.swing.JFrame;

public class PruebaLeerArchivoServidor
{
   public static void main( String args[] )
   {
      LeerArchivoServidor aplicacion = new LeerArchivoServidor();
      aplicacion.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
   } // fin de main
} // fin de la clase PruebaLeerArchivoServidor
```



Figura 24.4 | Clase de prueba para `LeerArchivoServidor`. (Parte 1 de 2).



Figura 24.4 | Clase de prueba para `LeerArchivoServidor`. (Parte 2 de 2).

llama al método `actionPerformed` (líneas 31 a 34). En la línea 33 se utiliza el método `getActionCommand` de `ActionEvent` para obtener la cadena que introdujo el usuario en el objeto `JTextField`, y pasa esa cadena al método utilitario `obtenerLaPagina` (líneas 61 a 74).

En la línea 65 se invoca el método **`setPage`** de `JEditorPane` para descargar el documento especificado por `ubicacion` y mostrarlo en el objeto `JEditorPane`. Si ocurre un error al descargar el documento, el método `setPage` lanza una excepción `IOException`. Además, si se especifica un URL inválido ocurre una excepción `MalformedURLException` (una subclase de `IOException`). Si el documento se carga correctamente, en la línea 66 se muestra la ubicación actual en `campoIntroducir`.

Por lo general, un documento HTML contiene **hipervínculos** (texto, imágenes o componentes de la GUI) que, cuando se hace clic sobre ellos, proporcionan un acceso rápido a otro documento en Web. Si un objeto `JEditorPane` contiene un documento HTML y el usuario hace clic en un hipervínculo, el objeto `JEditorPane` genera un evento **`HyperlinkEvent`** (paquete `javax.swing.event`) y notifica a todos los objetos **`HyperlinkListener`** (paquete `javax.swing.event`) registrados acerca de ese evento. En las líneas 42 a 53 se registra un objeto `HyperlinkListener` para manejar eventos `HyperlinkEvent`. Al ocurrir un evento `HyperlinkEvent`, el programa llama al método **`hyperlinkUpdate`** (líneas 46 a 51). En las líneas 48 y 49 se utiliza el método **`getEventType`** de `HyperlinkEvent` para determinar el tipo del evento `HyperlinkEvent`. La clase `HyperlinkEvent` contiene una clase anidada `public` llamada **`EventType`**, la cual declara tres objetos `EventType` estáticos que representan los tipos de eventos de hipervínculos. **`ACTIVATED`** indica que el usuario hizo clic en un hipervínculo para cambiar de página Web, **`ENTERED`** indica que el usuario movió el ratón sobre un hipervínculo y **`EXITED`** indica que el usuario alejó el ratón de un hipervínculo. Si un hipervínculo fue activado (`ACTIVATED`), en la línea 50 se utiliza el método **`getURL`** de `HyperlinkEvent` para obtener el `URL` representado por el hipervínculo. El método `toString` convierte el `URL` devuelto en una cadena que puede pasarse al método utilitario `obtenerLaPagina`.



Observación de apariencia visual 24.1

Un objeto `JEditorPane` genera eventos `HyperlinkEvent` solamente si no puede editarse.

24.4 Cómo establecer un servidor simple utilizando sockets de flujo

Los dos ejemplos descritos hasta ahora utilizan herramientas de red de alto nivel en Java para la comunicación entre las aplicaciones. En esos ejemplos no es responsabilidad del programador de Java establecer la conexión entre un cliente y un servidor. El primer programa dependió del navegador Web para comunicarse con un servidor Web. El segundo dependió de un objeto `JEditorPane` para realizar la conexión. En esta sección comenzaremos con nuestra discusión acerca de cómo crear sus propias aplicaciones que puedan comunicarse entre sí.

Para establecer un servidor simple en Java se requieren cinco pasos. El *paso 1* es crear un objeto **ServerSocket**. Una llamada al constructor de ServerSocket como:

```
ServerSocket servidor = new ServerSocket( numeroPuerto, longitudCola );
```

registra un **número de puerto** TCP disponible y específica el máximo número de clientes que pueden esperar para conectarse al servidor (es decir, la **longitud de la cola**). El número de puerto es utilizado por los clientes para localizar la aplicación servidor en el equipo servidor. A menudo, a esto se le conoce como **punto de negociación (handshake)**. Si la cola está llena, el servidor rechaza las conexiones de los clientes. El constructor establece el puerto en donde el servidor espera las conexiones de los clientes; a este proceso se le conoce como **enlazar el servidor al puerto**. Cada cliente pedirá conectarse con el servidor en este **puerto**. Sólo una aplicación puede enlazarse a un puerto específico en el servidor, en un momento dado.



Observación de ingeniería de software 24.1

Los números de puerto pueden ser entre 0 y 65,535. Algunos sistemas operativos reservan los números de puertos menores que 1024 para los servicios del sistema (como los servidores de e-mail y World Wide Web). Por lo general, estos puertos no deben especificarse como puertos de conexión en los programas de los usuarios. De hecho, algunos sistemas operativos requieren de privilegios de acceso especiales para enlazarse a los números de puerto menores que 1024.

Los programas administran cada conexión cliente mediante un objeto **Socket**. En el *paso 2*, el servidor escucha indefinidamente (o **bloquea**) para esperar a que un cliente trate de conectarse. Para escuchar una conexión de un cliente, el programa llama al método **accept** de ServerSocket, como se muestra a continuación:

```
Socket conexion = servidor.accept();
```

esta instrucción devuelve un objeto Socket cuando se establece la conexión con un cliente. El objeto Socket permite al servidor interactuar con el cliente. Las interacciones con el cliente ocurren realmente en un puerto del servidor distinto al del punto de negociación. De esta forma, el puerto especificado en el *paso 1* puede utilizarse nuevamente en un servidor con subprocesamiento múltiple, para aceptar otra conexión cliente. En la sección 24.8 demostraremos este concepto.

El *paso 3* es obtener los objetos OutputStream e InputStream que permiten al servidor comunicarse con el cliente, enviando y recibiendo bytes. El servidor envía información al cliente mediante un objeto OutputStream y recibe información del cliente mediante un objeto InputStream. El servidor invoca al método **getOutputStream** en el objeto Socket para obtener una referencia al objeto OutputStream del objeto Socket, e invoca al método **getInputStream** en el objeto Socket para obtener una referencia al objeto InputStream del objeto Socket.

Los objetos flujo pueden utilizarse para enviar o recibir bytes individuales, o secuencias de bytes, mediante el método write de OutputStream y el método read de InputStream, respectivamente. A menudo es útil enviar o recibir valores de tipos primitivos (como int y double) u objetos Serializable (como objetos String u otros tipos serializables), en vez de enviar bytes. En este caso podemos utilizar las técnicas del capítulo 14 para envolver otros tipos de flujos (como ObjectOutputStream y ObjectInputStream) alrededor de los objetos OutputStream e InputStream asociados con el objeto Socket. Por ejemplo,

```
ObjectInputStream entrada =
    new ObjectInputStream( conexion.getInputStream() );

ObjectOutputStream salida =
    new ObjectOutputStream( conexion.getOutputStream() );
```

Lo mejor de establecer estas relaciones es que, cualquier cosa que escriba el servidor en el objeto ObjectOutputStream se enviará mediante el objeto OutputStream y estará disponible en el objeto InputStream del cliente, y cualquier cosa que el cliente escriba en su objeto OutputStream (mediante su correspondiente objeto ObjectOutputStream) estará disponible a través del objeto InputStream del servidor. La transmisión de los datos a través de la red es un proceso transparente, y se maneja completamente mediante Java.

El *paso 4* es la fase de *procesamiento*, en la cual el servidor y el cliente se comunican a través de los objetos `OutputStream` e `InputStream`. En el *paso 5*, cuando se completa la transmisión, el servidor cierra la conexión invocando al método `close` en los flujos y en el objeto `Socket`.



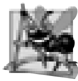
Observación de ingeniería de software 24.2

Con los sockets, la E/S de red es vista por los programas de Java como algo similar a un archivo de E/S secuencial. Los sockets ocultan al programador gran parte de la complejidad de la programación en red.



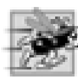
Observación de ingeniería de software 24.3

Mediante el subprocesamiento múltiple en Java, podemos crear servidores que utilicen esta característica y puedan administrar muchas conexiones simultáneas con muchos clientes. Esta arquitectura de servidor con subprocesamiento múltiple es precisamente lo que utilizan los servidores de red populares.



Observación de ingeniería de software 24.4

Un servidor con subprocesamiento múltiple puede tomar el `Socket` *devuelto por cada llamada al método* `accept`*, y puede crear un nuevo subproceso que administre la E/S de red a través de ese objeto* `Socket`*. Como alternativa, un servidor con subprocesamiento múltiple puede mantener una reserva de subprocesos (un conjunto de subprocesos ya existentes) listos para administrar la E/S de red a través de los nuevos objetos* `Socket`*, a medida que se vayan creando. En el capítulo 23 podrá consultar más información acerca del subprocesamiento múltiple.*



Tip de rendimiento 24.2

En los sistemas de alto rendimiento en los que la memoria es abundante, puede implementarse un servidor con subprocesamiento múltiple para crear una reserva de subprocesos que puedan asignarse rápidamente para manejar la E/S de red a través de cada nuevo `Socket`*, a medida que se vayan creando. Por lo tanto, cuando el servidor recibe una conexión, no necesita incurrir en la sobrecarga que se genera debido a la creación de los subprocesos. Cuando se cierra la conexión, el subproceso se devuelve a la reserva para su reutilización.*

24.5 Cómo establecer un cliente simple utilizando sockets de flujo

Para establecer un cliente simple en Java se requieren cuatro pasos. En el *paso 1* creamos un objeto `Socket` para conectarse al servidor. El constructor de `Socket` establece la conexión al servidor. Por ejemplo, la instrucción:

```
Socket conexion = new Socket( direccionServidor, puerto );
```

utiliza el constructor de `Socket` con dos argumentos: la dirección del servidor (*direccionServidor*) y el número de *puerto*. Si el intento de conexión es exitoso, esta instrucción devuelve un objeto `Socket`. Un intento de conexión fallido lanzará una instancia de una subclase de `IOException`, por lo que muchos programas simplemente atrapan a `IOException`. Una excepción `UnknownHostException` ocurre específicamente cuando el sistema no puede resolver la dirección del servidor especificada en la llamada al constructor de `Socket` en una dirección IP que corresponda.

En el *paso 2*, el cliente utiliza los métodos `getInputStream` y `getOutputStream` de la clase `Socket` para obtener referencias a los objetos `InputStream` y `OutputStream` de `Socket`. Como dijimos en la sección anterior, podemos utilizar las técnicas del capítulo 14 para envolver otros tipos de flujos alrededor de los objetos `InputStream` y `OutputStream` asociados con el objeto `Socket`. Si el servidor va a enviar información en el formato de los tipos actuales, el cliente debe recibir esa información en el mismo formato. Por lo tanto, si el servidor envía los valores con un objeto `ObjectOutputStream`, el cliente debe leer esos valores con un objeto `ObjectInputStream`.

El *paso 3* es la fase de procesamiento, en la cual el cliente y el servidor se comunican a través de los objetos `InputStream` y `OutputStream`. En el *paso 4*, el cliente cierra la conexión cuando se completa la transmisión, invocando al método `close` en los flujos y en el objeto `Socket`. El cliente debe determinar cuándo va a terminar el servidor de enviar información, de manera que pueda llamar al método `close` para cerrar la conexión del objeto `Socket`. Por ejemplo, el método `read` de `InputStream` devuelve el valor –1 cuando detecta el fin del flujo (lo que se conoce también como EOF: fin del archivo). Si se utiliza un objeto `ObjectInputStream` para leer información del servidor, se produce una excepción `EOFException` cuando el cliente trata de leer un valor de un flujo en el que se detectó el fin del flujo.

24.6 Interacción entre cliente/servidor mediante conexiones de socket de flujo

En las figuras 24.5 y 24.7 utilizamos sockets de flujo para demostrar una **aplicación cliente/servidor para conversar** simple. El servidor espera un intento de conexión por parte de un cliente. Cuando se conecta un cliente al servidor, la aplicación servidor envía un objeto `String` al cliente (recuerde que los objetos `String` son objetos `Serializable`), indicando que la conexión con el cliente fue exitosa. Después el cliente muestra el mensaje. Ambas aplicaciones cliente y servidor proporcionan campos de texto que permiten al usuario escribir un mensaje y enviarlo de una a otra aplicación. Cuando el cliente o el servidor envían la cadena `"TERMINAR"`, la conexión entre el cliente y el servidor termina. Después el servidor espera a que se conecte otro cliente. La declaración de la clase `Servidor` aparece en la figura 24.5. La declaración de la clase `Cliente` aparece en la figura 24.7. Las capturas de pantalla en las que se muestra la ejecución entre el cliente y el servidor aparecen como parte de la figura 24.7.

La clase `Servidor`

El constructor de `Servidor` (líneas 30 a 55) crea la GUI del servidor, la cual contiene un objeto `JTextField` y un objeto `JTextArea`. `Servidor` muestra su salida en el objeto `JTextArea`. Cuando se ejecuta el método `main` (líneas 7 a 12 de la figura 24.6) crea un objeto `Servidor`, especifica la operación de cierre predeterminada de la ventana y llama al método `ejecutarServidor` (declarado en las líneas 58 a 87).

El método `ejecutarServidor` configura el servidor para que reciba una conexión y procese una conexión a la vez. En la línea 62 se crea un objeto `ServerSocket` llamado `servidor`, para que espere las conexiones. El objeto `ServerSocket` escucha en el puerto `12345`, en espera de que un cliente se conecte. El segundo argumento para el constructor es el número de conexiones que pueden esperar en una cola para conectarse al servidor (`100` en este ejemplo). Si la cola está llena cuando un cliente trate de conectarse, el servidor rechazará la conexión.

```
1  // Fig. 24.5: Servidor.java
2  // Establece un servidor que recibe una conexión de un cliente, envía
3  // una cadena al cliente y cierra la conexión.
4  import java.io.EOFException;
5  import java.io.IOException;
6  import java.io.ObjectInputStream;
7  import java.io.ObjectOutputStream;
8  import java.net.ServerSocket;
9  import java.net.Socket;
10 import java.awt.BorderLayout;
11 import java.awt.event.ActionEvent;
12 import java.awt.event.ActionListener;
13 import javax.swing.JFrame;
14 import javax.swing.JScrollPane;
15 import javax.swing.JTextArea;
16 import javax.swing.JTextField;
17 import javax.swing.SwingUtilities;
18
19 public class Servidor extends JFrame
20 {
21    private JTextField campoIntroducir; // recibe como entrada un mensaje del usuario
22    private JTextArea areaPantalla; // muestra información al usuario
23    private ObjectOutputStream salida; // flujo de salida hacia el cliente
24    private ObjectInputStream entrada; // flujo de entrada del cliente
25    private ServerSocket servidor; // socket servidor
26    private Socket conexion; // conexión al cliente
27    private int contador = 1; // contador del número de conexiones
28
```

Figura 24.5 | Porción correspondiente al servidor de una conexión cliente/servidor con socket de flujo. (Parte 1 de 4).

```
   // establece la GUI
   public Servidor()
   {
      super( "Servidor" );

      campoIntroducir = new JTextField(); // crea objeto campoIntroducir
      campoIntroducir.setEditable( false );
      campoIntroducir.addActionListener(
         new ActionListener()
         {
            // envía un mensaje al cliente
            public void actionPerformed( ActionEvent evento )
            {
               enviarDatos( evento.getActionCommand() );
               campoIntroducir.setText( "" );
            } // fin del método actionPerformed
         } // fin de la clase interna anónima
      ); // fin de la llamada a addActionListener

      add( campoIntroducir, BorderLayout.NORTH );

      areaPantalla = new JTextArea(); // crea objeto areaPantalla
      add( new JScrollPane( areaPantalla ), BorderLayout.CENTER );

      setSize( 300, 150 ); // establece el tamaño de la ventana
      setVisible( true ); // muestra la ventana
   } // fin del constructor de Servidor

   // establece y ejecuta el servidor
   public void ejecutarServidor()
   {
      try // establece el servidor para que reciba conexiones; procesa las conexiones
      {
         servidor = new ServerSocket( 12345, 100 ); // crea objeto ServerSocket

         while ( true )
         {
            try
            {
               esperarConexion(); // espera una conexión
               obtenerFlujos(); // obtiene los flujos de entrada y salida
               procesarConexion(); // procesa la conexión
            } // fin de try
            catch ( EOFException excepcionEOF )
            {
               mostrarMensaje( "\nServidor termino la conexion" );
            } // fin de catch
            finally
            {
               cerrarConexion(); // cierra la conexión
               contador++;
            } // fin de finally
         } // fin de while
      } // fin de try
      catch ( IOException exepcionES )
      {
         exepcionES.printStackTrace();
      } // fin de catch
   } // fin del método ejecutarServidor
```

Figura 24.5 | Porción correspondiente al servidor de una conexión cliente/servidor con socket de flujo. (Parte 2 de 4).

```

    // espera a que llegue una conexión, después muestra información sobre ésta
    private void esperarConexion() throws IOException
    {
       mostrarMensaje( "Esperando una conexion\n" );
       conexion = servidor.accept(); // permite al servidor aceptar la conexión
       mostrarMensaje( "Conexion " + contador + " recibida de: " +
          conexion.getInetAddress().getHostName() );
    } // fin del método esperarConexion

    // obtiene flujos para enviar y recibir datos
    private void obtenerFlujos() throws IOException
    {
       // establece el flujo de salida para los objetos
       salida = new ObjectOutputStream( conexion.getOutputStream() );
       salida.flush(); // vacía el búfer de salida para enviar información del encabezado

       // establece el flujo de entrada para los objetos
       entrada = new ObjectInputStream( conexion.getInputStream() );

       mostrarMensaje( "\nSe obtuvieron los flujos de E/S\n" );
    } // fin del método obtenerFlujos

    // procesa la conexión con el cliente
    private void procesarConexion() throws IOException
    {
       String mensaje = "Conexion exitosa";
       enviarDatos( mensaje ); // envía mensaje de conexión exitosa

       // habilita campoIntroducir para que el usuario del servidor pueda enviar mensajes
       setTextFieldEditable( true );

       do // procesa los mensajes enviados desde el cliente
       {
          try // lee el mensaje y lo muestra en pantalla
          {
             mensaje = ( String ) entrada.readObject(); // lee el nuevo mensaje
             mostrarMensaje( "\n" + mensaje ); // muestra el mensaje
          } // fin de try
          catch ( ClassNotFoundException excepcionClaseNoEncontrada )
          {
             mostrarMensaje( "\nSe recibio un tipo de objeto desconocido" );
          } // fin de catch

       } while ( !mensaje.equals( "CLIENTE>>> TERMINAR" ) );
    } // fin del método procesarConexion

    // cierra flujos y socket
    private void cerrarConexion()
    {
       mostrarMensaje( "\nTerminando conexion\n" );
       setTextFieldEditable( false ); // deshabilita campoIntroducir

       try
       {
          salida.close(); // cierra flujo de salida
          entrada.close(); // cierra flujo de entrada
          conexion.close(); // cierra el socket
       } // fin de try
```

Figura 24.5 | Porción correspondiente al servidor de una conexión cliente/servidor con socket de flujo. (Parte 3 de 4).

```
147          catch ( IOException exepcionES )
148          {
149             exepcionES.printStackTrace();
150          } // fin de catch
151       } // fin del método cerrarConexion
152
153       // envía el mensaje al cliente
154       private void enviarDatos( String mensaje )
155       {
156          try // envía objeto al cliente
157          {
158             salida.writeObject( "SERVIDOR>>> " + mensaje );
159             salida.flush(); // envía toda la salida al cliente
160             mostrarMensaje( "\nSERVIDOR>>> " + mensaje );
161          } // fin de try
162          catch ( IOException exepcionES )
163          {
164             areaPantalla.append( "\nError al escribir objeto" );
165          } // fin de catch
166       } // fin del método enviarDatos
167
168       // manipula areaPantalla en el subproceso despachador de eventos
169       private void mostrarMensaje( final String mensajeAMostrar )
170       {
171          SwingUtilities.invokeLater(
172             new Runnable()
173             {
174                public void run() // actualiza areaPantalla
175                {
176                   areaPantalla.append( mensajeAMostrar ); // adjunta el mensaje
177                } // fin del método run
178             } // fin de la clase interna anónima
179          ); // fin de la llamada a SwingUtilities.invokeLater
180       } // fin del método mostrarMensaje
181
182       // manipula a campoIntroducir en el subproceso despachador de eventos
183       private void setTextFieldEditable( final boolean editable )
184       {
185          SwingUtilities.invokeLater(
186             new Runnable()
187             {
188                public void run() // establece la propiedad de edición de campoIntroducir
189                {
190                   campoIntroducir.setEditable( editable );
191                } // fin del método
192             } // fin de la clase interna
193          ); // fin de la llamada a SwingUtilities.invokeLater
194       } // fin del método setTextFieldEditable
195    } // fin de la clase Servidor
```

Figura 24.5 | Porción correspondiente al servidor de una conexión cliente/servidor con socket de flujo. (Parte 4 de 4).



Error común de programación 24.1

Al especificar un puerto que ya esté en uso, o especificar un número de puerto incorrecto al crear un objeto `ServerSocket`, se produce una excepción `BindException`.

En la línea 68 se hace una llamada al método `esperarConexion` (declarado en las líneas 90 a 96) para esperar la conexión de un cliente. Una vez establecida la conexión, en la línea 69 se hace una llamada al método

```
// Fig. 24.6: PruebaServidor.java
// Prueba la aplicación Servidor.
import javax.swing.JFrame;

public class PruebaServidor
{
   public static void main( String args[] )
   {
      Servidor aplicacion = new Servidor(); // crea el servidor
      aplicacion.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
      aplicacion.ejecutarServidor(); // ejecuta la aplicación servidor
   } // fin de main
} // fin de la clase PruebaServidor
```

Figura 24.6 | Clase de prueba para `Servidor`.

`obtenerFlujos` (declarado en las líneas 99 a 109) para obtener las referencias a los flujos para la conexión. En la línea 70 se hace una llamada al método `procesarConexion` (declarado en las líneas 112 a 133) para enviar el mensaje de conexión inicial al cliente, y para procesar todos los mensajes que se reciban del cliente. El bloque `finally` (líneas 76 a 80) finaliza la conexión del cliente llamando al método `cerrarConexion` (líneas 136 a 151), incluso aunque haya ocurrido una excepción. El método `mostrarMensaje` (líneas 169 a 180) es llamado desde estos métodos para utilizar el subproceso despachador de eventos para mostrar los mensajes en el objeto `JTextArea` de la aplicación.

En el método `esperarConexion` (líneas 90 a 96) se utiliza el método `accept` de `ServerSocket` (línea 93) para esperar una conexión de un cliente. Al ocurrir una conexión, el objeto `Socket` resultante se asigna a `conexion`. El método `accept` realiza un bloqueo hasta que se reciba una conexión (es decir, el subproceso en el que se haga la llamada a `accept` detiene su ejecución hasta que un cliente se conecte). En las líneas 94 y 95 se imprime en pantalla el nombre del equipo host que realizó la conexión. El método **`getInetAddress`** de `Socket` devuelve un objeto **`InetAddress`** (paquete `java.net`), el cual contiene información acerca del equipo cliente. El método **`getHostName`** de `InetAddress` devuelve el nombre de host del equipo cliente. Por ejemplo, hay una dirección IP (`127.0.0.1`) y nombre de host (`localhost`) especiales, que son útiles para probar aplicaciones de red en su equipo local [a ésta también se le conoce como **dirección de bucle local (loopback)**]. Si se hace una llamada a `getHostName` en un objeto `InetAddress` que contenga `127.0.0.1`, el nombre de host correspondiente que devuelve el método es `localhost`.

El método `obtenerFlujos` (líneas 99 a 109) obtiene las referencias a los flujos de `Socket` y los utiliza para inicializar un objeto `ObjectOutputStream` (línea 102) y un objeto `ObjectInputStream` (línea 106), respectivamente. Observe la llamada al método **`flush`** de `ObjectInputStream` en la línea 103. Esta instrucción hace que el objeto `ObjectOutputStream` en el servidor envíe un **encabezado de flujo** al objeto `ObjectInputStream` correspondiente del cliente. El encabezado de flujo contiene información como la versión de la serialización de objetos que se va a utilizar para enviar los objetos. Esta información es requerida por el objeto `ObjectInputStream`, para que pueda prepararse para recibir estos objetos de manera correcta.



Observación de ingeniería de software 24.5

Al utilizar un objeto `ObjectOutputStream` y un objeto `ObjectInputStream` para enviar y recibir datos a través de una conexión de red, siempre debe crear primero el objeto `ObjectOutputStream` y vaciar (mediante el método `flush`) el flujo, de manera que el objeto `ObjectInputStream` pueda prepararse para recibir los datos. Esto se requiere sólo en las aplicaciones de red que se comunican utilizando a `ObjectOutputStream` y `ObjectInputStream`.



Tip de rendimiento 24.3

Por lo general, los componentes de entrada y salida de una computadora son mucho más lentos que su memoria. Los búferes de salida se utilizan comúnmente para incrementar la eficiencia de una aplicación, al enviar mayores cantidades de datos con menos frecuencia, con lo cual se reduce el número de veces que una aplicación accede a los componentes de entrada y salida de la computadora.

En la línea 114 del método procesarConexion (líneas 112 a 133) se hace una llamada al método enviarDatos para enviar la cadena "SERVIDOR>>> Conexión exitosa" al cliente. El ciclo de las líneas 120 a 132 se ejecuta hasta que el servidor recibe el mensaje "CLIENTE>>> TERMINAR." En la línea 124 se utiliza el método readObject de ObjectInputStream para leer un objeto String del cliente. En la línea 125 se invoca el método mostrarMensaje para anexar el mensaje al objeto JTextArea.

Cuando termina la transmisión, el método procesarConexion regresa y el programa llama al método cerrarConexion (líneas 136 a 151) para cerrar los flujos asociados con el objeto Socket, y para cerrar también a ese objeto Socket. Después, el servidor espera el siguiente intento de conexión por parte de un cliente, continuando con la línea 68 al principio del ciclo while.

Cuando el usuario de la aplicación servidor introduce una cadena en el campo de texto y oprime la tecla *Intro*, el programa llama al método actionPerformed (líneas 40 a 44), el cual lee la cadena del campo de texto y llama al método utilitario enviarDatos (líneas 154 a 166) para enviar la cadena al cliente. El método enviarDatos escribe el objeto, vacía el búfer de salida y anexa la misma cadena al área de texto en la ventana del servidor. No es necesario invocar a mostrarMensaje para modificar el área de texto aquí, ya que el método enviarDatos es llamado desde un manejador de eventos; por lo tanto, enviarDatos se ejecuta como parte del subproceso despachador de eventos.

Observe que el objeto Servidor recibe una conexión, la procesa, cierra la conexión y espera a la siguiente conexión. Un escenario más apropiado sería un objeto Servidor que recibe una conexión, la configura para procesarla como un subproceso de ejecución separado, e inmediatamente se pone en espera de nuevas conexiones. Los subprocesos separados que procesan conexiones existentes pueden seguir ejecutándose, mientras que el Servidor se concentra en nuevas peticiones de conexión. Esto hace al servidor más eficiente, ya que pueden procesarse varias peticiones de clientes en forma concurrente. En la sección 24.8 mostraremos el uso de un servidor con subprocesamiento múltiple.

La clase **Cliente**

Al igual que la clase Servidor, el constructor (líneas 29 a 56) de la clase Cliente (figura 24.7) crea la GUI de la aplicación (un objeto JTextField y un objeto JTextArea). Cliente muestra su salida en el área de texto. Cuando se ejecuta el método main (líneas 7 a 19 de la figura 24.8), se crea una instancia de la clase Cliente, se especifica la operación de cierre predeterminada de la ventana y se hace una llamada al método ejecutarCliente (declarado en las líneas 59 a 79). En este ejemplo, usted puede ejecutar el cliente desde cualquier computadora en Internet, y especificar la dirección IP o nombre de host del equipo servidor como un argumento de línea de comandos para el programa. Por ejemplo, el comando:

```
java Cliente 192.168.1.15
```

intenta realizar una conexión al objeto Servidor en la computadora que tiene la dirección IP 192.168.1.15.

El método ejecutarCliente (líneas 59 a 79) de la clase Cliente establece la conexión con el servidor, procesa los mensajes recibidos del servidor y cierra la conexión cuando termina la comunicación. En la línea 63 se hace una llamada al método conectarAlServidor (declarado en las líneas 82 a 92) para realizar la conexión. Después de realizar la conexión, en la línea 64 se hace una llamada al método obtenerFlujos (declarado en las líneas 95 a 105) para obtener las referencias a los objetos flujo del objeto Socket. Después, en la línea 65 se hace una llamada al método procesarConexion (declarado en las líneas 108 a 126) para recibir y mostrar los mensajes enviados por el servidor. En el bloque finally (líneas 75 a 78) se hace una llamada al método cerrarConexion (líneas 129 a 144) para cerrar los flujos y el objeto Socket, incluso aunque haya ocurrido una excepción. El método mostrarMensaje (líneas 162 a 173) es llamado desde estos métodos para utilizar el subproceso despachador de eventos para mostrar los mensajes en el área de texto de la aplicación.

```
1  // Fig. 24.7: Cliente.java
2  // Cliente que lee y muestra la información que se envía desde un Servidor.
3  import java.io.EOFException;
4  import java.io.IOException;
5  import java.io.ObjectInputStream;
```

Figura 24.7 | Porción correspondiente al cliente, de una conexión de sockets de flujo entre un cliente y un servidor. (Parte 1 de 5).

```
import java.io.ObjectOutputStream;
import java.net.InetAddress;
import java.net.Socket;
import java.awt.BorderLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JFrame;
import javax.swing.JScrollPane;
import javax.swing.JTextArea;
import javax.swing.JTextField;
import javax.swing.SwingUtilities;

public class Cliente extends JFrame
{
   private JTextField campoIntroducir; // introduce la información del usuario
   private JTextArea areaPantalla; // muestra la información al usuario
   private ObjectOutputStream salida; // flujo de salida hacia el servidor
   private ObjectInputStream entrada; // flujo de entrada del servidor
   private String mensaje = ""; // mensaje del servidor
   private String servidorChat; // aloja al servidor para esta aplicación
   private Socket cliente; // socket para comunicarse con el servidor

   // inicializa el objeto servidorChat y establece la GUI
   public Cliente( String host )
   {
      super( "Cliente" );

      servidorChat = host; // establece el servidor al que se conecta este cliente

      campoIntroducir = new JTextField(); // crea objeto campoIntroducir
      campoIntroducir.setEditable( false );
      campoIntroducir.addActionListener(
         new ActionListener()
         {
            // envía el mensaje al servidor
            public void actionPerformed( ActionEvent evento )
            {
               enviarDatos( evento.getActionCommand() );
               campoIntroducir.setText( "" );
            } // fin del método actionPerformed
         } // fin de la clase interna anónima
      ); // fin de la llamada a addActionListener

      add( campoIntroducir, BorderLayout.NORTH );

      areaPantalla = new JTextArea(); // crea objeto areaPantalla
      add( new JScrollPane( areaPantalla ), BorderLayout.CENTER );

      setSize( 300, 150 ); // establece el tamaño de la ventana
      setVisible( true ); // muestra la ventana
   } // fin del constructor de Cliente

   // se conecta al servidor y procesa los mensajes que éste envía
   public void ejecutarCliente()
   {
      try // se conecta al servidor, obtiene flujos, procesa la conexión
      {
         conectarAlServidor(); // crea un objeto Socket para hacer la conexión
```

Figura 24.7 | Porción correspondiente al cliente, de una conexión de sockets de flujo entre un cliente y un servidor. (Parte 2 de 5).

```
         obtenerFlujos(); // obtiene los flujos de entrada y salida
         procesarConexion(); // procesa la conexión
      } // fin de try
      catch ( EOFException excepcionEOF )
      {
         mostrarMensaje( "\nCliente termino la conexion" );
      } // fin de catch
      catch ( IOException excepcionES )
      {
         excepcionES.printStackTrace();
      } // fin de catch
      finally
      {
         cerrarConexion(); // cierra la conexión
      } // fin de finally
   } // fin del método ejecutarCliente

   // se conecta al servidor
   private void conectarAlServidor() throws IOException
   {
      mostrarMensaje( "Intentando realizar conexion\n" );

      // crea objeto Socket para hacer conexión con el servidor
      cliente = new Socket( InetAddress.getByName( servidorChat ), 12345 );

      // muestra la información de la conexión
      mostrarMensaje( "Conectado a: " +
         cliente.getInetAddress().getHostName() );
   } // fin del método conectarAlServidor

   // obtiene flujos para enviar y recibir datos
   private void obtenerFlujos() throws IOException
   {
      // establece flujo de salida para los objetos
      salida = new ObjectOutputStream( cliente.getOutputStream() );
      salida.flush(); // vacía el búfer de salida para enviar información de encabezado

      // establece flujo de entrada para los objetos
      entrada = new ObjectInputStream( cliente.getInputStream() );

      mostrarMensaje( "\nSe obtuvieron los flujos de E/S\n" );
   } // fin del método obtenerFlujos

   // procesa la conexión con el servidor
   private void procesarConexion() throws IOException
   {
      // habilita campoIntroducir para que el usuario cliente pueda enviar mensajes
      establecerCampoEditable( true );

      do // procesa los mensajes que se envían desde el servidor
      {
         try // lee el mensaje y lo muestra
         {
            mensaje = ( String ) entrada.readObject(); // lee nuevo mensaje
            mostrarMensaje( "\n" + mensaje ); // muestra el mensaje
         } // fin de try
         catch ( ClassNotFoundException excepcionClaseNoEncontrada )
         {
```

Figura 24.7 | Porción correspondiente al cliente, de una conexión de sockets de flujo entre un cliente y un servidor. (Parte 3 de 5).

```
            mostrarMensaje( "nSe recibio un tipo de objeto desconocido" );
         } // fin de catch

      } while ( !mensaje.equals( "SERVIDOR>>> TERMINAR" ) );
   } // fin del método procesarConexion

   // cierra flujos y socket
   private void cerrarConexion()
   {
      mostrarMensaje( "\nCerrando conexion" );
      establecerCampoEditable( false ); // deshabilita campoIntroducir

      try
      {
         salida.close(); // cierra el flujo de salida
         entrada.close(); // cierra el flujo de entrada  1
         cliente.close(); // cierra el socket
      } // fin de try
      catch ( IOException excepcionES )
      {
         excepcionES.printStackTrace();
      } // fin de catch
   } // fin del método cerrarConexion

   // envía un mensaje al servidor
   private void enviarDatos( String mensaje )
   {
      try // envía un objeto al servidor
      {
         salida.writeObject( "CLIENTE>>> " + mensaje );
         salida.flush(); // envía todos los datos a la salida
         mostrarMensaje( "\nCLIENTE>>> " + mensaje );
      } // fin de try
      catch ( IOException excepcionES )
      {
         areaPantalla.append( "\nError al escribir objeto" );
      } // fin de catch
   } // fin del método enviarDatos

   // manipula el objeto areaPantalla en el subproceso despachador de eventos
   private void mostrarMensaje( final String mensajeAMostrar )
   {
      SwingUtilities.invokeLater(
         new Runnable()
         {
            public void run() // actualiza objeto areaPantalla
            {
               areaPantalla.append( mensajeAMostrar );
            } // fin del método run
         } // fin de la clase interna anónima
      ); // fin de la llamada a SwingUtilities.invokeLater
   } // fin del método mostrarMensaje

   // manipula a campoIntroducir en el subproceso despachador de eventos
   private void establecerCampoEditable( final boolean editable )
   {
      SwingUtilities.invokeLater(
         new Runnable()
```

Figura 24.7 | Porción correspondiente al cliente, de una conexión de sockets de flujo entre un cliente y un servidor. (Parte 4 de 5).

```
180           {
181              public void run() // establece la propiedad de edición de campoIntroducir
182              {
183                 campoIntroducir.setEditable( editable );
184              } // fin del método run
185           } // fin de la clase interna anónima
186        ); // fin de la llamada a SwingUtilities.invokeLater
187     } // fin del método establecerCampoEditable
188  } // fin de la clase Cliente
```

Figura 24.7 | Porción correspondiente al cliente, de una conexión de sockets de flujo entre un cliente y un servidor. (Parte 5 de 5).

El método `conectarAlServidor` (líneas 82 a 92) crea un objeto `Socket` llamado `cliente` (línea 87) para establecer una conexión. Este método pasa dos argumentos al constructor de `Socket`: la dirección IP del equipo servidor y el número de puerto (12345) en donde la aplicación servidor está esperando las conexiones de los clientes. En el primer argumento, el método `static` **`getByName`** de `InetAddress` devuelve un objeto `InetAddress` que contiene la dirección IP especificada como argumento en la línea de comandos para la aplicación (o `127.0.0.1` si no se especifican argumentos en la línea de comandos). El método `getByName` puede recibir una cadena que contiene la dirección IP actual, o el nombre de host del servidor. El primer argumento también podría haberse escrito de otras formas. Para la dirección 127.0.0.1 de localhost, el primer argumento podría especificarse mediante una de las siguientes expresiones:

```
InetAddress.getByName( "localhost" )
InetAddress.getLocalHost()
```

```
// Fig. 24.8: PruebaCliente.java
// Prueba la clase Cliente.
import javax.swing.JFrame;

public class PruebaCliente
{
   public static void main( String args[] )
   {
      Cliente aplicacion; // declara la aplicación cliente

      // si no hay argumentos de línea de comandos
      if ( args.length == 0 )
         aplicacion = new Cliente( "127.0.0.1" ); // se conecta a localhost
      else
         aplicacion = new Cliente( args[ 0 ] ); // usa args para conectarse

      aplicacion.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
      aplicacion.ejecutarCliente(); // ejecuta la aplicación cliente
   } // fin de main
} // fin de la clase PruebaCliente
```

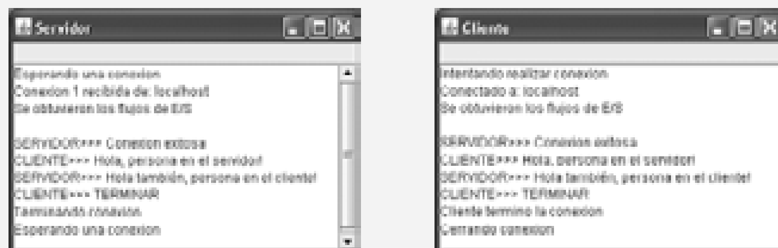


Figura 24.8 | Clase que prueba a `Cliente`.

Además, hay versiones del constructor de Socket que reciben una cadena para la dirección IP o nombre de host. El primer argumento podía haberse especificado como "127.0.0.1" o "localhost". Optamos por demostrar la relación cliente/servidor mediante una conexión entre aplicaciones que se ejecutan en la misma computadora (localhost). Por lo general, este primer argumento sería la dirección IP de otra computadora. El objeto InetAddress para otra computadora se puede obtener si especificamos la dirección IP o nombre de host de la computadora como argumento para el método getByName de InetAddress. El segundo argumento del constructor de Socket es el número de puerto del servidor. Este número debe concordar con el número de puerto en el que el servidor esté esperando las conexiones (al cual se le conoce como el punto de negociación, o handshake). Una vez que se realiza la conexión, en las líneas 90 y 91 se muestra un mensaje en el área de texto, indicando el nombre del equipo servidor al cual se conectó el cliente.

El objeto Cliente utiliza un objeto ObjectOutputStream para enviar datos al servidor, y un objeto ObjectInputStream para recibir datos del servidor. El método obtenerFlujos (líneas 95 a 105) crea los objetos ObjectOutputStream y ObjectInputStream que utilizan los flujos asociados con el socket del cliente.

El método procesarConexion (líneas 108 a 126) contiene un ciclo que se ejecuta hasta que el cliente recibe el mensaje "SERVIDOR>>> TERMINAR". En la línea 117 se lee un objeto String del servidor. En la línea 118 se invoca el método mostrarMensaje para anexar el mensaje al área de texto.

Cuando la transmisión termina, el método cerrarConexion (líneas 129 a 144) cierra los flujos y el objeto Socket.

Cuando el usuario de la aplicación cliente introduce una cadena en el campo de texto y oprime la tecla *Intro*, el programa llama al método actionPerformed (líneas 41 a 45) para leer la cadena e invoca al método utilitario enviarDatos (líneas 147 a 159) para enviar la cadena al servidor. El método enviarDatos escribe el objeto, vacía el búfer de salida y anexa la misma cadena al objeto JTextArea en la ventana del cliente. Una vez más, no es necesario invocar al método utilitario mostrarMensaje para modificar el área de texto aquí, ya que el método enviarDatos es llamado desde un manejador de eventos.

24.7 Interacción entre cliente/servidor sin conexión mediante datagramas

Hasta este punto hemos hablado sobre la transmisión basada en flujos, orientada a la conexión. Ahora consideremos la **transmisión sin conexión mediante datagramas**.

La transmisión orientada a la conexión es como el sistema telefónico en el que usted marca y recibe una conexión al teléfono de la persona con la que usted desea comunicarse. La conexión se mantiene todo el tiempo que dure su llamada telefónica, incluso aunque usted no esté hablando.

La transmisión sin conexión mediante datagramas es un proceso más parecido a la manera en que el correo se transporta mediante el servicio postal. Si un mensaje extenso no cabe en un sobre, usted lo divide en varias piezas separadas que coloca en sobres separados, numerados en forma secuencial. Cada una de las cartas se envía entonces por correo al mismo tiempo. Las cartas podrían llegar en orden, sin orden o tal vez no llegarían (aunque el último caso es raro, suele ocurrir). La persona en el extremo receptor debe reensamblar las piezas del mensaje en orden secuencial, antes de tratar de interpretarlo. Si su mensaje es lo suficientemente pequeño como para caber en un sobre, no tiene que preocuparse por el problema de que el mensaje esté "fuera de secuencia", pero aún existe la posibilidad de que su mensaje no llegue. Una diferencia entre los datagramas y el correo postal es que pueden llegar duplicados de datagramas al equipo receptor.

En las figuras 24.9 a 24.12 utilizamos datagramas para enviar paquetes de información mediante el Protocolo de datagramas de usuario (UDP) entre una aplicación cliente y una aplicación servidor. En la aplicación Cliente (figura 24.11), el usuario escribe un mensaje en un campo de texto y oprime *Intro*. El programa convierte el mensaje en un arreglo byte y lo coloca en un paquete de datagramas que se envía al servidor. El Servidor (figura 24.9) recibe el paquete y muestra la información que contiene, después lo repite (**echo**) de vuelta al cliente. Cuando el cliente recibe el paquete, muestra la información que contiene.

La clase Servidor

La clase Servidor (figura 24.9) declara dos objetos **DatagramPacket** que son utilizados por el servidor para enviar y recibir información, y un objeto **DatagramSocket** que envía y recibe estos paquetes. El constructor de Servidor (líneas 19 a 37) crea la interfaz gráfica de usuario en la que se mostrarán los paquetes de información. A continuación, en la línea 30 se crea el objeto DatagramSocket en un bloque try. En la línea 30 se utiliza el

constructor de `DatagramSocket` que recibe un argumento entero para el número de puerto (`5000` en este ejemplo) para enlazar el servidor a un puerto en donde pueda recibir paquetes de los clientes. Los objetos `Cliente` que envían paquetes a este objeto `Servidor` especifican el mismo número de puerto en los paquetes que envían. Si el constructor de **`DatagramSocket`** no puede enlazar el objeto `DatagramSocket` al puerto especificado, se lanza una excepción **`SocketException`**.

```java
// Fig. 24.9: Servidor.java
// Servidor que recibe y envía paquetes desde/hacia un cliente.
import java.io.IOException;
import java.net.DatagramPacket;
import java.net.DatagramSocket;
import java.net.SocketException;
import java.awt.BorderLayout;
import javax.swing.JFrame;
import javax.swing.JScrollPane;
import javax.swing.JTextArea;
import javax.swing.SwingUtilities;

public class Servidor extends JFrame
{
   private JTextArea areaPantalla; // muestra los paquetes recibidos
   private DatagramSocket socket; // socket para conectarse al cliente

   // establece la GUI y el objeto DatagramSocket
   public Servidor()
   {
      super( "Servidor" );

      areaPantalla = new JTextArea(); // crea objeto areaPantalla
      add( new JScrollPane( areaPantalla ), BorderLayout.CENTER );
      setSize( 400, 300 ); // establece el tamaño de la ventana
      setVisible( true ); // muestra la ventana

      try // crea objeto DatagramSocket para enviar y recibir paquetes
      {
         socket = new DatagramSocket( 5000 );
      } // fin de try
      catch ( SocketException excepcionSocket )
      {
         excepcionSocket.printStackTrace();
         System.exit( 1 );
      } // fin de catch
   } // fin del constructor de Servidor

   // espera a que lleguen los paquetes, muestra los datos y repite el paquete al cliente
   public void esperarPaquetes()
   {
      while ( true )
      {
         try // recibe el paquete, muestra su contenido, devuelve una copia al cliente
         {
            byte datos[] = new byte[ 100 ]; // establece un paquete
            DatagramPacket paqueteRecibir =
               new DatagramPacket( datos, datos.length );

            socket.receive( paqueteRecibir ); // espera a recibir el paquete

            // muestra la información del paquete recibido
```

Figura 24.9 | Lado servidor de la computación cliente/servidor sin conexión mediante datagramas. (Parte 1 de 2).

```
53                mostrarMensaje( "\nPaquete recibido:" +
54                   "\nDe host: " + paqueteRecibir.getAddress() +
55                   "\nPuerto host: " + paqueteRecibir.getPort() +
56                   "\nLongitud: " + paqueteRecibir.getLength() +
57                   "\nContiene:\n\t" + new String( paqueteRecibir.getData(),
58                      0, paqueteRecibir.getLength() ) );
59
60                enviarPaqueteAlCliente( paqueteRecibir ); // envía el paquete al cliente
61             } // fin de try
62             catch ( IOException excepcionES )
63             {
64                mostrarMensaje( excepcionES.toString() + "\n" );
65                excepcionES.printStackTrace();
66             } // fin de catch
67          } // fin de while
68       } // fin del método esperarPaquetes
69
70       // repite el paquete al cliente
71       private void enviarPaqueteAlCliente( DatagramPacket paqueteRecibir )
72          throws IOException
73       {
74          mostrarMensaje( "\n\nRepitiendo datos al cliente..." );
75
76          // crea paquete para enviar
77          DatagramPacket paqueteEnviar = new DatagramPacket(
78             paqueteRecibir.getData(), paqueteRecibir.getLength(),
79             paqueteRecibir.getAddress(), paqueteRecibir.getPort() );
80
81          socket.send( paqueteEnviar ); // envía paquete al cliente
82          mostrarMensaje( "Paquete enviado\n" );
83       } // fin del método enviarPaqueteAlCliente
84
85       // manipula objeto areaPantalla en el subproceso despachador de eventos
86       private void mostrarMensaje( final String mensajeAMostrar )
87       {
88          SwingUtilities.invokeLater(
89             new Runnable()
90             {
91                public void run() // actualiza areaPantalla
92                {
93                   areaPantalla.append( mensajeAMostrar ); // muestra mensaje
94                } // fin del método run
95             } // fin de la clase interna anónima
96          ); // fin de la llamada a SwingUtilities.invokeLater
97       } // fin del método mostrarMensaje
98    } // fin de la clase Servidor
```

Figura 24.9 | Lado servidor de la computación cliente/servidor sin conexión mediante datagramas. (Parte 2 de 2).



Error común de programación 24.2

Al especificar un puerto que ya esté en uso, o especificar un número de puerto incorrecto al crear un objeto `DatagramSocket`, se produce una excepción `SocketException`.

El método `esperarPaquetes` (líneas 40 a 68) de la clase `Servidor` utiliza un ciclo infinito para esperar a que lleguen los paquetes al `Servidor`. En las líneas 47 y 48 se crea un objeto `DatagramPacket`, en el cual puede almacenarse un paquete de información que se haya recibido. El constructor de `DatagramPacket` para este fin recibe dos argumentos: un arreglo `byte` en el cual se almacenarán los datos y la longitud del arreglo. En la línea 50 se utiliza el método **`receive`** de `DatagramSocket` para esperar a que llegue un paquete al `Servidor`. El método

```java
// Fig. 24.10: PruebaServidor.java
// Prueba la clase Servidor.
import javax.swing.JFrame;

public class PruebaServidor
{
   public static void main( String args[] )
   {
      Servidor aplicacion = new Servidor(); // crea el servidor
      aplicacion.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
      aplicacion.esperarPaquetes(); // ejecuta la aplicación servidor
   } // fin de main
} // fin de la clase PruebaServidor
```



Ventana del `Servidor` después de recibir el paquete de datos del `Cliente`

Figura 24.10 | Clase que prueba el `Servidor`.

`receive` hace un bloqueo hasta que llega el paquete y después lo almacena en su argumento `DatagramPacket`. Este método lanza una excepción `IOException` si ocurre un error al recibir un paquete.

Cuando llega un paquete, en las líneas 53 a 58 se hace una llamada al método `mostrarMensaje` (declarado en las líneas 86 a 97) para anexar el contenido del paquete al área de texto. El método **`getAddress`** de `DatagramPacket` (línea 54) devuelve un objeto `InetAddress` que contiene el nombre de host del equipo desde el que se envió el paquete. El método **`getPort`** (línea 55) devuelve un entero que especifica el número de puerto a través del que el equipo host enviará el paquete. El método **`getLength`** (línea 56) devuelve un entero que representa el número de bytes de datos que se enviaron. El método **`getData`** (línea 57) devuelve un arreglo `byte` que contiene los datos. En las líneas 57 y 58 se inicializa un objeto `String` mediante el uso de un constructor de tres argumentos que recibe un arreglo `byte`, el desplazamiento y la longitud. Después, este objeto `String` se anexa al texto que se mostrará en pantalla.

Después de mostrar un paquete, en la línea 60 se hace una llamada al método `enviarPaqueteAlCliente` (declarado en las líneas 71 a 83) para crear un nuevo paquete y enviarlo al cliente. En las líneas 77 a 79 se crea un objeto `DatagramPacket` y se le pasan cuatro argumentos a su constructor. El primer argumento especifica el arreglo `byte` a enviar. El segundo especifica el número de bytes a enviar. El tercer argumento especifica la dirección de Internet del equipo cliente a donde se va a enviar el paquete. El cuarto argumento especifica el puerto en el que el cliente espera recibir los paquetes. En la línea 81 se envía el paquete a través de la red. El método **`send`** de `DatagramSocket` lanza una excepción `IOException` si ocurre un error al enviar un paquete.

La clase `Cliente`

La clase `Cliente` (figura 24.11) funciona de manera similar a la clase `Servidor`, excepto que el `Cliente` envía paquetes sólo cuando el usuario escribe un mensaje en un campo de texto y oprime la tecla *Intro*. Cuando esto ocurre, el programa llama al método `actionPerformed` (líneas 32 a 57), el cual convierte la cadena que introdujo el usuario en un arreglo `byte` (línea 41). En las líneas 44 y 45 se crea un objeto `DatagramPacket` y se inicializa

con el arreglo `byte`, la longitud de la cadena introducida por el usuario, la dirección IP a la que se enviará el paquete (`InetAddress.getLocalHost()` en este ejemplo) y el número de puerto en el que el `Servidor` espera los paquetes (5000 en este ejemplo). En la línea 47 se envía el paquete. Observe que el cliente en este ejemplo debe saber que el servidor está recibiendo paquetes en el puerto 5000; de no ser así, el servidor no los recibirá.

```
1  // Fig. 24.11: Cliente.java
2  // Cliente que envía y recibe paquetes desde/hacia un servidor.
3  import java.io.IOException;
4  import java.net.DatagramPacket;
5  import java.net.DatagramSocket;
6  import java.net.InetAddress;
7  import java.net.SocketException;
8  import java.awt.BorderLayout;
9  import java.awt.event.ActionEvent;
10 import java.awt.event.ActionListener;
11 import javax.swing.JFrame;
12 import javax.swing.JScrollPane;
13 import javax.swing.JTextArea;
14 import javax.swing.JTextField;
15 import javax.swing.SwingUtilities;
16
17 public class Cliente extends JFrame
18 {
19    private JTextField campoIntroducir; // para introducir mensajes
20    private JTextArea areaPantalla; // para mostrar mensajes
21    private DatagramSocket socket; // socket para conectarse al servidor
22
23    // establece la GUI y el objeto DatagramSocket
24    public Cliente()
25    {
26       super( "Cliente" );
27
28       campoIntroducir = new JTextField( "Escriba aqui el mensaje" );
29       campoIntroducir.addActionListener(
30          new ActionListener()
31          {
32             public void actionPerformed( ActionEvent evento )
33             {
34                try // crea y envía un paquete
35                {
36                   // obtiene mensaje del campo de texto
37                   String mensaje = evento.getActionCommand();
38                   areaPantalla.append( "\nEnviando paquete que contiene: " +
39                      mensaje + "\n" );
40
41                   byte datos[] = mensaje.getBytes(); // convierte en bytes
42
43                   // crea objeto sendPacket
44                   DatagramPacket paqueteEnviar = new DatagramPacket( datos,
45                      datos.length, InetAddress.getLocalHost(), 5000 );
46
47                   socket.send( paqueteEnviar ); // envía el paquete
48                   areaPantalla.append( "Paquete enviado\n" );
49                   areaPantalla.setCaretPosition(
50                      areaPantalla.getText().length() );
51                } // fin de try
52                catch ( IOException excepcionES )
```

Figura 24.11 | Lado cliente de la computación cliente/servidor sin conexión mediante datagramas. (Parte 1 de 3).

```
               {
                  mostrarMensaje( excepcionES.toString() + "\n" );
                  excepcionES.printStackTrace();
               } // fin de catch
            } // fin de actionPerformed
         } // fin de la clase interna
      ); // fin de la llamada a addActionListener

      add( campoIntroducir, BorderLayout.NORTH );

      areaPantalla = new JTextArea();
      add( new JScrollPane( areaPantalla ), BorderLayout.CENTER );

      setSize( 400, 300 ); // establece el tamaño de la ventana
      setVisible( true ); // muestra la ventana

      try // crea objeto DatagramSocket para enviar y recibir paquetes
      {
         socket = new DatagramSocket();
      } // fin de try
      catch ( SocketException excepcionSocket )
      {
         excepcionSocket.printStackTrace();
         System.exit( 1 );
      } // fin de catch
   } // fin del constructor de Cliente

   // espera a que lleguen los paquetes del servidor, muestra el contenido de éstos
   public void esperarPaquetes()
   {
      while ( true )
      {
         try // recibe paquete y muestra su contenido
         {
            byte datos[] = new byte[ 100 ]; // establece el paquete
            DatagramPacket paqueteRecibir = new DatagramPacket(
               datos, datos.length );

            socket.receive( paqueteRecibir ); // espera el paquete

            // muestra el contenido del paquete
            mostrarMensaje( "\nPaquete recibido:" +
               "\nDe host: " + paqueteRecibir.getAddress() +
               "\nPuerto host: " + paqueteRecibir.getPort() +
               "\nLongitud: " + paqueteRecibir.getLength() +
               "\nContiene:\n\t" + new String( paqueteRecibir.getData(),
                  0, paqueteRecibir.getLength() ) );
         } // fin de try
         catch ( IOException excepcion )
         {
            mostrarMensaje( excepcion.toString() + "\n" );
            excepcion.printStackTrace();
         } // fin de catch
      } // fin de while
   } // fin del método esperarPaquetes

   // manipula objeto areaPantalla en el subproceso despachador de eventos
   private void mostrarMensaje( final String mensajeAMostrar )
   {
```

Figura 24.11 | Lado cliente de la computación cliente/servidor sin conexión mediante datagramas. (Parte 2 de 3).

```
112         SwingUtilities.invokeLater(
113            new Runnable()
114            {
115               public void run() // actualiza objeto areaPantalla
116               {
117                  areaPantalla.append( mensajeAMostrar );
118               } // fin del método run
119            } // fin de la clase interna
120         ); // fin de la llamada a SwingUtilities.invokeLater
121      } // fin del método mostrarMensaje
122   }  // fin de la clase Cliente
```

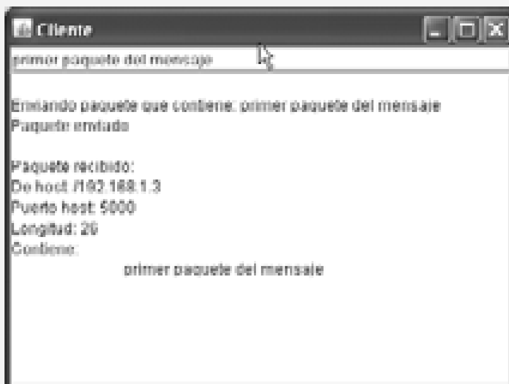
Figura 24.11 | Lado cliente de la computación cliente/servidor sin conexión mediante datagramas. (Parte 3 de 3).

Observe que la llamada al constructor de `DatagramSocket` (línea 71) en esta aplicación no especifica ningún argumento. Este constructor sin argumentos permite a la computadora seleccionar el siguiente número de puerto disponible para el objeto `DatagramSocket`. El cliente no necesita un número de puerto específico, ya que el servidor recibe el número de puerto del cliente como parte de cada objeto `DatagramPacket` enviado por el cliente. Por lo tanto, el servidor puede enviar paquetes de vuelta al mismo equipo y número de puerto desde el cual haya recibido un paquete de información.

El método `esperarPaquetes` (líneas 81 a 107) de la clase `Cliente` utiliza un ciclo infinito para esperar a que lleguen los paquetes del servidor. En la línea 91 se hace un bloqueo hasta que llegue un paquete. Esto no impide al usuario enviar un paquete, ya que los eventos de la GUI se manejan en el subproceso despachador de eventos. Sólo impide que el ciclo `while` continúe ejecutándose, hasta que llegue un paquete al `Cliente`. Cuando llega un paquete, en la línea 91 se almacena este paquete en `paqueteRecibir`, y en las líneas 94 a 99 se hace una llamada al método `mostrarMensaje` (declarado en las líneas 110 a 121) para mostrar el contenido del paquete en el área de texto.

```
// Fig. 24.12: PruebaCliente.java
// Prueba la clase Cliente.
import javax.swing.JFrame;

public class PruebaCliente
{
   public static void main( String args[] )
   {
      Cliente aplicacion = new Cliente(); // crea el cliente
      aplicacion.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
      aplicacion.esperarPaquetes(); // ejecuta la aplicación cliente
   } // fin de main
}  // fin de la clase PruebaCliente
```



Ventana del lado del `Cliente` después de enviar el paquete del `Servidor` y recibirlo de regreso

Figura 24.12 | Clase que prueba el `Cliente`.

24.8 Juego de Tres en raya (Gato) tipo cliente/servidor, utilizando un servidor con subprocesamiento múltiple

En esta sección presentaremos el popular juego de Tres en raya (Gato o Tic-Tac-Toe), que implementaremos mediante el uso de las técnicas cliente/servidor con sockets de flujo. El programa consiste en una aplicación `TresEnRaya` (figuras 24.13 y 24.14) que permite a dos aplicaciones `ClienteTresEnRaya` (figuras 24.15 y 24.16) conectarse al servidor y jugar Tres en raya. En la figura 24.17 se muestran pantallas de ejemplo de la salida de este programa.

La clase ServidorTresEnRaya

A medida que el `ServidorTresEnRaya` recibe la conexión de cada cliente, crea una instancia de la clase interna `Jugador` (líneas 182 a 304 de la figura 24.13) para procesar al cliente en un subproceso separado. Estos subprocesos permiten a los clientes jugar en forma independiente. El primer cliente que se conecta al servidor es el jugador X y el segundo es el jugador O. El jugador X realiza el primer movimiento. El servidor mantiene la información acerca del tablero, para poder determinar si el movimiento de un jugador es válido o inválido.

```
1  // Fig. 24.13: ServidorTresEnRaya.java
2  // Esta clase mantiene un juego de Tres en raya para dos clientes.
3  import java.awt.BorderLayout;
4  import java.net.ServerSocket;
5  import java.net.Socket;
6  import java.io.IOException;
7  import java.util.Formatter;
8  import java.util.Scanner;
9  import java.util.concurrent.ExecutorService;
10 import java.util.concurrent.Executors;
11 import java.util.concurrent.locks.Lock;
12 import java.util.concurrent.locks.ReentrantLock;
13 import java.util.concurrent.locks.Condition;
14 import javax.swing.JFrame;
15 import javax.swing.JTextArea;
16 import javax.swing.SwingUtilities;
17
18 public class ServidorTresEnRaya extends JFrame
19 {
20    private String[] tablero = new String[ 9 ]; // tablero de tres en raya
21    private JTextArea areaSalida; // para imprimir los movimientos en pantalla
22    private Jugador[] jugadores; // arreglo de objetos Jugador
23    private ServerSocket servidor; // socket servidor para conectarse con los clientes
24    private int jugadorActual; // lleva la cuenta del jugador que sigue en turno
25    private final static int JUGADOR_X = 0; // constante para el primer jugador
26    private final static int JUGADOR_O = 1; // constante para el segundo jugador
27    private final static String[] MARCAS = { "X", "O" }; // arreglo de marcas
28    private ExecutorService ejecutarJuego; // ejecuta a los jugadores
29    private Lock bloqueoJuego; // para bloquear el juego y estar sincronizado
30    private Condition otroJugadorConectado; // para esperar al otro jugador
31    private Condition turnoOtroJugador; // para esperar el turno del otro jugador
32
33    // establece servidor de tres en raya y GUI para mostrar mensajes en pantalla
34    public ServidorTresEnRaya()
35    {
36       super( "Servidor de Tres en raya" ); // establece el título de la ventana
37
38       // crea objeto ExecutorService con un subproceso para cada jugador
39       ejecutarJuego = Executors.newFixedThreadPool( 2 );
40       bloqueoJuego = new ReentrantLock(); // crea bloqueo para el juego
```

Figura 24.13 | Lado servidor del programa Tres en raya cliente/servidor. (Parte 1 de 6).

```java

      // variable de condición para los dos jugadores conectados
      otroJugadorConectado = bloqueoJuego.newCondition();

      // variable de condición para el turno del otro jugador
      turnoOtroJugador = bloqueoJuego.newCondition();

      for ( int i = 0; i < 9; i++ )
         tablero[ i ] = new String( "" ); // crea tablero de tres en raya
      jugadores = new Jugador[ 2 ]; // crea arreglo de jugadores
      jugadorActual = JUGADOR_X; // establece el primer jugador como el jugador actual

      try
      {
         servidor = new ServerSocket( 12345, 2 ); // establece objeto ServerSocket
      } // fin de try
      catch ( IOException excepcionES )
      {
         excepcionES.printStackTrace();
         System.exit( 1 );
      } // fin de catch

      areaSalida = new JTextArea(); // crea objeto JTextArea para mostrar la salida
      add( areaSalida, BorderLayout.CENTER );
      areaSalida.setText( "Servidor esperando conexiones\n" );

      setSize( 300, 300 ); // establece el tamaño de la ventana
      setVisible( true ); // muestra la ventana
   } // fin del constructor de ServidorTresEnRaya

   // espera dos conexiones para poder jugar
   public void execute()
   {
      // espera a que se conecte cada cliente
      for ( int i = 0; i < jugadores.length; i++ )
      {
         try // espera la conexión, crea el objeto Jugador, inicia objeto Runnable
         {
            jugadores[ i ] = new Jugador( servidor.accept(), i );
            ejecutarJuego.execute( jugadores[ i ] ); // ejecuta el objeto Runnable jugador
         } // fin de try
         catch ( IOException excepcionES )
         {
            excepcionES.printStackTrace();
            System.exit( 1 );
         } // fin de catch
      } // fin de for

      bloqueoJuego.lock(); // bloquea el juego para avisar al subproceso del jugador X

      try
      {
         jugadores[ JUGADOR_X ].establecerSuspendido( false ); // continúa el jugador X
         otroJugadorConectado.signal(); // despierta el subproceso del jugador X
      } // fin de try
      finally
      {
         bloqueoJuego.unlock(); // desbloquea el juego después de avisar al jugador X
      } // fin de finally
```

Figura 24.13 | Lado servidor del programa Tres en raya cliente/servidor. (Parte 2 de 6).

```
   } // fin del método execute

   // muestra un mensaje en objeto areaSalida
   private void mostrarMensaje( final String mensajeAMostrar )
   {
      // muestra un mensaje del subproceso de ejecución despachador de eventos
      SwingUtilities.invokeLater(
         new Runnable()
         {
            public void run() // actualiza el objeto areaSalida
            {
               areaSalida.append( mensajeAMostrar ); // agrega el mensaje
            } // fin del método run
         } // fin de la clase interna
      ); // fin de la llamada a SwingUtilities.invokeLater
   } // fin del método mostrarMensaje

   // determina si el movimiento es válido
   public boolean validarYMover( int ubicacion, int jugador )
   {
      // mientras no sea el jugador actual, debe esperar su turno
      while ( jugador != jugadorActual )
      {
         bloqueoJuego.lock(); // bloquea el juego para esperar a que el otro jugador
         haga su movmiento

         try
         {
            turnoOtroJugador.await(); // espera el turno de jugador
         } // fin de try
         catch ( InterruptedException excepcion )
         {
            excepcion.printStackTrace();
         } // fin de catch
         finally
         {
            bloqueoJuego.unlock(); // desbloquea el juego después de esperar
         } // fin de finally
      } // fin de while

      // si la ubicación no está ocupada, realiza el movimiento
      if ( !estaOcupada( ubicacion ) )
      {
         tablero[ ubicacion ] = MARCAS[ jugadorActual ]; // establece el movimiento en
         el tablero
         jugadorActual = ( jugadorActual + 1 ) % 2; // cambia el jugador

         // deja que el nuevo jugador sepa que se realizó un movimiento
         jugadores[ jugadorActual ].otroJugadorMovio( ubicacion );

         bloqueoJuego.lock(); // bloquea el juego para indicar al otro jugador que
         realice su movimiento

         try
         {
            turnoOtroJugador.signal(); // indica al otro jugador que debe continuar
         } // fin de try
         finally
         {
```

Figura 24.13 | Lado servidor del programa Tres en raya cliente/servidor. (Parte 3 de 6).

```
            bloqueoJuego.unlock(); // desbloquea el juego despues de avisar
         } // fin de finally

         return true; // notifica al jugador que el movimiento fue válido
      } // fin de if
      else // el movimiento no fue válido
         return false; // notifica al jugador que el movimiento fue inválido
   } // fin del método validarYMover

   // determina si la ubicación está ocupada
   public boolean estaOcupada( int ubicacion )
   {
      if ( tablero[ ubicacion ].equals( MARCAS[ JUGADOR_X ] ) ||
         tablero [ ubicacion ].equals( MARCAS[ JUGADOR_O ] ) )
         return true; // la ubicación está ocupada
      else
         return false; // la ubicación no está ocupada
   } // fin del método estaOcupada

   // coloca código en este método para determinar si terminó el juego
   public boolean seTerminoJuego()
   {
      return false; // esto se deja como ejercicio
   } // fin del método seTerminoJuego

   // la clase interna privada Jugador maneja a cada Jugador como objeto Runnable
   private class Jugador implements Runnable
   {
      private Socket conexion; // conexión con el cliente
      private Scanner entrada; // entrada del cliente
      private Formatter salida; // salida al cliente
      private int numeroJugador; // rastrea cuál jugador es el actual
      private String marca; // marca para este jugador
      private boolean suspendido = true; // indica si el subproceso está suspendido

      // establece el subproceso Jugador
      public Jugador( Socket socket, int numero )
      {
         numeroJugador = numero; // almacena el número de este jugador
         marca = MARCAS[ numeroJugador ]; // especifica la marca del jugador
         conexion = socket; // almacena socket para el cliente

         try // obtiene los flujos del objeto Socket
         {
            entrada = new Scanner( conexion.getInputStream() );
            salida = new Formatter( conexion.getOutputStream() );
         } // fin de try
         catch ( IOException excepcionES )
         {
            excepcionES.printStackTrace();
            System.exit( 1 );
         } // fin de catch
      } // fin del constructor de Jugador

      // envía mensaje que indica que el otro jugador hizo un movimiento
      public void otroJugadorMovio( int ubicacion )
      {
         salida.format( "El oponente realizo movimiento\n" );
         salida.format( "%d\n", ubicacion ); // envía la ubicación del movimiento
```

Figura 24.13 | Lado servidor del programa Tres en raya cliente/servidor. (Parte 4 de 6).

```
         salida.flush(); // vacía la salida
      } // fin del método otroJugadorMovio

      // controla la ejecución del subproceso
      public void run()
      {
         // envía al cliente su marca (X o O), procesa los mensajes del cliente
         try
         {
            mostrarMensaje( "Jugador " + marca + " conectado\n" );
            salida.format( "%s\n", marca ); // envía la marca del jugador
            salida.flush(); // vacía la salida

            // si es el jugador X, espera a que llegue el otro jugador
            if ( numeroJugador == JUGADOR_X )
            {
               salida.format( "%s\n%s", "Jugador X conectado",
                  "Esperando al otro jugador\n" );
               salida.flush(); // vacía la salida

               bloqueoJuego.lock(); // bloquea el juego para esperar al segundo jugador

               try
               {
                  while( suspendido )
                  {
                     otroJugadorConectado.await(); // espera al jugador O
                  } // fin de while
               } // fin de try
               catch ( InterruptedException excepcion )
               {
                  excepcion.printStackTrace();
               } // fin de catch
               finally
               {
                  bloqueoJuego.unlock(); // desbloquea el juego después del segundo jugador
               } // fin de finally

               // envía un mensaje que indica que el otro jugador se conectó
               salida.format( "El otro jugador se conecto. Ahora es su turno.\n" );
               salida.flush(); // vacía la salida
            } // fin de if
            else
            {
               salida.format( "El jugador O se conecto, por favor espere\n" );
               salida.flush(); // vacía la salida
            } // fin de else

            // mientras el juego no termine
            while ( !seTerminoJuego() )
            {
               int ubicacion = 0; // inicializa la ubicación del movimiento

               if ( entrada.hasNext() )
                  ubicacion = entrada.nextInt(); // obtiene la ubicación del movimiento

               // comprueba si el movimiento es válido
               if ( validarYMover( ubicacion, numeroJugador ) )
```

Figura 24.13 | Lado servidor del programa Tres en raya cliente/servidor. (Parte 5 de 6).

```
273                  {
274                     mostrarMensaje( "\nubicacion: " + ubicacion );
275                     salida.format( "Movimiento valido.\n" ); // notifica al cliente
276                     salida.flush(); // vacía la salida
277                  } // fin de if
278                  else // el movimiento fue inválido
279                  {
280                     salida.format( "Movimiento invalido, intente de nuevo\n" );
281                     salida.flush(); // vacía la salida
282                  } // fin de else
283               } // fin de while
284            } // fin de try
285            finally
286            {
287               try
288               {
289                  conexion.close(); // cierra la conexión con el cliente
290               } // fin de try
291               catch ( IOException excepcionES )
292               {
293                  excepcionES.printStackTrace();
294                  System.exit( 1 );
295               } // fin de catch
296            } // fin de finally
297         } // fin del método run
298
299         // establece si se suspende el subproceso o no
300         public void establecerSuspendido( boolean estado )
301         {
302            suspendido = estado; // establece el valor de suspendido
303         } // fin del método establecerSuspendido
304      } // fin de la clase Jugador
305   } // fin de la clase ServidorTresEnRaya
```

Figura 24.13 | Lado servidor del programa Tres en raya cliente/servidor. (Parte 6 de 6).

Vamos a empezar con una discusión sobre el lado servidor del juego de Tres en raya. Al ejecutarse la aplicación `ServidorTresEnRaya`, el método `main` (líneas 7 a 12 de la figura 24.14) crea un objeto `ServidorTresEnRaya` llamado `aplicacion`. El constructor (líneas 34 a 69 de la figura 24.13) trata de establecer un objeto `ServerSocket`. Si tiene éxito, el programa muestra la ventana del servidor y después `main` invoca al método `execute` (líneas 72 a 100) de `ServidorTresEnRaya`. El método `execute` itera dos veces, haciendo un bloqueo en la línea 79 cada vez que espera la conexión de un cliente. Cuando un cliente se conecta, en la línea 79 se crea un nuevo objeto `Jugador` para administrar la conexión como un subproceso separado, y en la línea 80 se ejecuta el objeto `Jugador` en la reserva de subprocesos `ejecutarJuego`.

Cuando el `ServidorTresEnRaya` crea a un `Jugador`, el constructor de `Jugador` (líneas 192 a 208) recibe el objeto `Socket` que representa la conexión con el cliente y obtiene los flujos de entrada y salida asociados. En la línea 201 se crea un objeto `Formatter` (vea el capítulo 28) y se envuelve alrededor del flujo de salida del socket. El método `run` de `Jugador` (líneas 219 a 297) controla la información que se envía al cliente y la información que se recibe del cliente. Primero, pasa al cliente el carácter que va a colocar en el tablero cuando se haga un movimiento (línea 225). En la línea 226 se hace una llamada al método `flush` de `Formatter` para forzar esta salida al cliente. En la línea 241 se suspende el subproceso del jugador X cuando empieza a ejecutarse, ya que el jugador X podrá realizar movimientos sólo hasta después de que el jugador O se conecte.

Una vez que se conecta el jugador O se puede empezar a jugar, y el método `run` empieza a ejecutar su estructura `while` (líneas 264 a 283). En cada iteración de este ciclo se lee un entero (línea 269) que representa la ubicación en donde el cliente desea colocar una marca, y en la línea 272 se invoca al método `validarYMover` (declarado en las líneas 118 a 163) de `ServidorTresEnRaya` para comprobar el movimiento. Si el movimiento es válido, en la línea 275 se envía un mensaje al cliente, indicando que el movimiento fue válido. En caso contrario,

en la línea 280 se envía un mensaje al cliente indicando que el movimiento fue inválido. El programa mantiene las ubicaciones en el tablero como números del 0 al 8 (del 0 al 2 para la primera fila, del 3 al 5 para la segunda y del 6 al 8 para la tercera).

```
// Fig. 24.14: PruebaServidorTresEnRaya.java
// Prueba el ServidorTresEnRaya.
import javax.swing.JFrame;

public class PruebaServidorTresEnRaya
{
   public static void main( String args[] )
   {
      ServidorTresEnRaya aplicacion = new ServidorTresEnRaya();
      aplicacion.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
      aplicacion.execute();
   } // fin de main
} // fin de la clase PruebaServidorTresEnRaya
```



Figura 24.14 | Clase que prueba el servidor de Tres en raya.

El método `validarYMover` (líneas 118 a 163 en la clase `ServidorTresEnRaya`) sólo permite que se mueva un jugador en un momento dado, con lo cual se evita que ambos jugadores modifiquen la información de estado del juego al mismo tiempo. Si el `Jugador` que trata de validar un movimiento no es el jugador actual (es decir, el que puede hacer un movimiento), ese `Jugador` se coloca en estado de *espera* hasta que sea su turno para realizar un movimiento. Si la ubicación para el movimiento que se está validando ya está ocupada en el tablero, `validarY-Mover` devuelve `false`. En caso contrario, el servidor coloca una marca para el jugador en su representación local del tablero (línea 142), notifica al otro objeto `Jugador` (línea 146) que se ha realizado un movimiento (para que se pueda enviar un mensaje al cliente), invoca al método `signal` (línea 152) para que el `Jugador` en espera (si hay uno) pueda validar un movimiento y devuelve `true` (línea 159) para indicar que el movimiento es válido.

La clase `ClienteTresEnRaya`

Cada aplicación `ClienteTresEnRaya` (figura 24.15) mantiene su propia versión de GUI del tablero de Tres en raya en el que se muestra el estado del juego. Los clientes pueden colocar una marca solamente en un cuadro vacío en el tablero. La clase interna `Cuadro` (líneas 205 a 262 de la figura 24.15) implementa a cada uno de los nueve cuadros en el tablero. Cuando un objeto `ClienteTresEnRaya` comienza a ejecutarse, crea un objeto `JTextArea` en el cual se muestran los mensajes del servidor, junto con una representación del tablero en la que se muestran nueve objetos `Cuadro`. El método `iniciarCliente` (líneas 80 a 100) abre una conexión con el servidor y obtiene los flujos de entrada y salida asociados del objeto `Socket`. En las líneas 85 y 86 se realiza una conexión con el servidor. La clase `ClienteTresEnRaya` implementa a la interfaz `Runnable`, por lo que un subproceso separado

puede leer los mensajes del servidor. Este método permite al usuario interactuar con el tablero (en el subproceso despachador de eventos) mientras espera mensajes del servidor. Después de establecer la conexión con el servidor, en la línea 99 se ejecuta el cliente con el objeto `ExecutorService` llamado `trabajador`. El método `run` (líneas 103 a 126) controla el subproceso de ejecución separado. Primero, el método lee el carácter de la marca (X u O) del servidor (línea 105), después itera continuamente (líneas 121 a 125) y lee los mensajes del servidor (línea 124). Cada mensaje se pasa al método `procesarMensaje` (líneas 129 a 156) para su procesamiento.

```
1  // Fig. 24.15: ClienteTresEnRaya.java
2  // Cliente que permite a un usuario jugar Tres en raya con otro a través de una red.
3  import java.awt.BorderLayout;
4  import java.awt.Dimension;
5  import java.awt.Graphics;
6  import java.awt.GridLayout;
7  import java.awt.event.MouseAdapter;
8  import java.awt.event.MouseEvent;
9  import java.net.Socket;
10 import java.net.InetAddress;
11 import java.io.IOException;
12 import javax.swing.JFrame;
13 import javax.swing.JPanel;
14 import javax.swing.JScrollPane;
15 import javax.swing.JTextArea;
16 import javax.swing.JTextField;
17 import javax.swing.SwingUtilities;
18 import java.util.Formatter;
19 import java.util.Scanner;
20 import java.util.concurrent.Executors;
21 import java.util.concurrent.ExecutorService;
22
23 public class ClienteTresEnRaya extends JFrame implements Runnable
24 {
25    private JTextField campoId; // campo de texto para mostrar la marca del jugador
26    private JTextArea areaPantalla; // objeto JTextArea para mostrar la salida
27    private JPanel panelTablero; // panel para el tablero de tres en raya
28    private JPanel panel2; // panel que contiene el tablero
29    private Cuadro tablero[][]; // tablero de tres en raya
30    private Cuadro cuadroActual; // el cuadro actual
31    private Socket conexion; // conexión con el servidor
32    private Scanner entrada; // entrada del servidor
33    private Formatter salida; // salida al servidor
34    private String hostTresEnRaya; // nombre de host para el servidor
35    private String miMarca; // la marca de este cliente
36    private boolean miTurno; // determina de qué cliente es el turno
37    private final String MARCA_X = "X"; // marca para el primer cliente
38    private final String MARCA_O = "O"; // marca para el segundo cliente
39
40    // establece la interfaz de usuario y el tablero
41    public ClienteTresEnRaya( String host )
42    {
43       hostTresEnRaya = host; // establece el nombre del servidor
44       areaPantalla = new JTextArea( 4, 30 ); // establece objeto JTextArea
45       areaPantalla.setEditable( false );
46       add( new JScrollPane( areaPantalla ), BorderLayout.SOUTH );
47
48       panelTablero = new JPanel(); // establece panel para los cuadros en el tablero
49       panelTablero.setLayout( new GridLayout( 3, 3, 0, 0 ) );
50
```

Figura 24.15 | Lado cliente del programa Tres en raya cliente/servidor. (Parte 1 de 5).

```
      tablero = new Cuadro[ 3 ][ 3 ]; // crea el tablero

      // itera a través de las filas en el tablero
      for ( int fila = 0; fila < tablero.length; fila++ )
      {
         // itera a través de las columnas en el tablero
         for ( int columna = 0; columna < tablero[ fila ].length; columna++ )
         {
            // crea un cuadro
            tablero[ fila ][ columna ] = new Cuadro( ' ', fila * 3 + columna );
            panelTablero.add( tablero[ fila ][ columna ] ); // agrega el cuadro
         } // fin de for interior
      } // fin de for exterior

      campoId = new JTextField(); // establece campo de texto
      campoId.setEditable( false );
      add( campoId, BorderLayout.NORTH );

      panel2 = new JPanel(); // establece el panel que contiene a panelTablero
      panel2.add( panelTablero, BorderLayout.CENTER ); // agrega el panel del
      tablero
      add( panel2, BorderLayout.CENTER ); // agrega el panel contenedor

      setSize( 325, 225 ); // establece el tamaño de la ventana
      setVisible( true ); // muestra la ventana

      iniciarCliente();
   } // fin del constructor de ClienteTresEnRaya

   // inicia el subproceso cliente
   public void iniciarCliente()
   {
      try // se conecta al servidor, obtiene los flujos e inicia subproceso de salida
      {
         // realiza conexión con el servidor
         conexion = new Socket(
            InetAddress.getByName( hostTresEnRaya ), 12345 );

         // obtiene flujos para entrada y salida
         entrada = new Scanner( conexion.getInputStream() );
         salida = new Formatter( conexion.getOutputStream() );
      } // fin de try
      catch ( IOException excepcionES )
      {
         excepcionES.printStackTrace();
      } // fin de catch

      // crea e inicia subproceso trabajador para este cliente
      ExecutorService trabajador = Executors.newFixedThreadPool( 1 );
      trabajador.execute( this ); // ejecuta el cliente
   } // fin del método iniciarCliente

   // subproceso de control que permite la actualización continua de areaPantalla
   public void run()
   {
      miMarca = entrada.nextLine(); // obtiene la marca del jugador (X o O)

      SwingUtilities.invokeLater(
         new Runnable()
```

Figura 24.15 | Lado cliente del programa Tres en raya cliente/servidor. (Parte 2 de 5).

```
         {
            public void run()
            {
               // muestra la marca del jugador
               campoId.setText( "Usted es el jugador \"" + miMarca + "\"" );
            } // fin del método run
         } // fin de la clase interna anónima
      ); // fin de la llamada a SwingUtilities.invokeLater

      miTurno = ( miMarca.equals( MARCA_X ) ); // determina si es turno del cliente

      // recibe los mensajes que se envían al cliente y los imprime en pantalla
      while ( true )
      {
         if ( entrada.hasNextLine() )
            procesarMensaje( entrada.nextLine() );
      } // fin de while
   } // fin del método run

   // procesa los mensajes recibidos por el cliente
   private void procesarMensaje( String mensaje )
   {
      // ocurrió un movimiento válido
      if ( mensaje.equals( "Movimiento valido." ) )
      {
         mostrarMensaje( "Movimiento valido, por favor espere.\n" );
         establecerMarca( cuadroActual, miMarca ); // establece marca en el cuadro
       } // fin de if
      else if ( mensaje.equals( "Movimiento invalido, intente de nuevo" ) )
      {
         mostrarMensaje( mensaje + "\n" ); // muestra el movimiento inválido
         miTurno = true; // sigue siendo turno de este cliente
      } // fin de else if
      else if ( mensaje.equals( "El oponente realizo movimiento" ) )
      {
         int ubicacion = entrada.nextInt(); // obtiene la ubicación del movimiento
         entrada.nextLine(); // salta nueva línea después de la ubicación int
         int fila = ubicacion / 3; // calcula la fila
         int columna = ubicacion % 3; // calcula la columna

         establecerMarca( tablero[ fila ][ columna ],
            ( miMarca.equals( MARCA_X ) ? MARCA_O : MARCA_X ) ); // marca el movimiento
         mostrarMensaje( "El oponente hizo un movimiento. Ahora es su turno.\n" );
         miTurno = true; // ahora es turno de este cliente
      } // fin de else if
      else
         mostrarMensaje( mensaje + "\n" ); // muestra el mensaje
   } // fin del método procesarMensaje

   // manipula el objeto areaSalida en el subproceso despachador de eventos
   private void mostrarMensaje( final String mensajeAMostrar )
   {
      SwingUtilities.invokeLater(
         new Runnable()
         {
            public void run()
            {
               areaPantalla.append( mensajeAMostrar ); // actualiza la salida
            } // fin del método run
```

Figura 24.15 | Lado cliente del programa Tres en raya cliente/servidor. (Parte 3 de 5).

```
         } // fin de la clase interna
      ); // fin de la llamada a SwingUtilities.invokeLater
   } // fin del método mostrarMensaje

   // método utilitario para establecer una marca en el tablero, en el subproceso despachador de eventos
   private void establecerMarca( final Cuadro cuadroAMarcar, final String marca )
   {
      SwingUtilities.invokeLater(
         new Runnable()
         {
            public void run()
            {
               cuadroAMarcar.establecerMarca( marca ); // establece la marca en el cuadro
            } // fin del método run
         } // fin de la clase interna anónima
      ); // fin de la llamada a SwingUtilities.invokeLater
   } // fin del método establecerMarca

   // envía un mensaje al servidor, indicando el cuadro en el que se hizo clic
   public void enviarCuadroClic( int ubicacion )
   {
      // si es mi turno
      if ( miTurno )
      {
         salida.format( "%d\n", ubicacion ); // envía la ubicación al servidor
         salida.flush();
         miTurno = false; // ya no es mi turno
      } // fin de if
   } // fin del método enviarCuadroClic

   // establece el cuadro actual
   public void establecerCuadroActual( Cuadro cuadro )
   {
      cuadroActual = cuadro; // asigna el argumento al cuadro actual
   } // fin del método establecerCuadroActual

   // clase interna privada para los cuadros en el tablero
   private class Cuadro extends JPanel
   {
      private String marca; // marca a dibujar en este cuadro
      private int ubicacion; // ubicacion del cuadro

      public Cuadro( String marcaCuadro, int ubicacionCuadro )
      {
         marca = marcaCuadro; // establece la marca para este cuadro
         ubicacion = ubicacionCuadro; // establece la ubicación de este cuadro

         addMouseListener(
            new MouseAdapter()
            {
               public void mouseReleased( MouseEvent e )
               {
                  establecerCuadroActual( Cuadro.this ); // establece el cuadro actual

                  // envía la ubicación de este cuadro
                  enviarCuadroClic( obtenerUbicacionCuadro() );
               } // fin del método mouseReleased
```

Figura 24.15 | Lado cliente del programa Tres en raya cliente/servidor. (Parte 4 de 5).

```
            } // fin de la clase interna anónima
         ); // fin de la llamada a addMouseListener
      } // fin del constructor de Cuadro

      // devuelve el tamaño preferido del objeto Cuadro
      public Dimension getPreferredSize()
      {
         return new Dimension( 30, 30 ); // devuelve el tamaño preferido
      } // fin del método getPreferredSize

      // devuelve el tamaño mínimo del objeto Cuadro
      public Dimension getMinimumSize()
      {
         return getPreferredSize(); // devuelve el tamaño preferido
      } // fin del método getMinimumSize

      // establece la marca para el objeto Cuadro
      public void establecerMarca( String nuevaMarca )
      {
         marca = nuevaMarca; // establece la marca del cuadro
         repaint(); // vuelve a pintar el cuadro
      } // fin del método establecerMarca

      // devuelve la ubicación del objeto Cuadro
      public int obtenerUbicacionCuadro()
      {
         return ubicacion; // devuelve la ubicación del cuadro
      } // fin del método obtenerUbicacionCuadro

      // dibuja el objeto Cuadro
      public void paintComponent( Graphics g )
      {
         super.paintComponent( g );

         g.drawRect( 0, 0, 29, 29 ); // dibuja el cuadro
         g.drawString( marca, 11, 20 ); // dibuja la marca
      } // fin del método paintComponent
   } // fin de la clase interna Cuadro
} // fin de la clase ClienteTresEnRaya
```

Figura 24.15 | Lado cliente del programa Tres en raya cliente/servidor. (Parte 5 de 5).

```
// Fig. 24.16: PruebaClienteTresEnRaya.java
// Prueba la clase ClienteTresEnRaya.
import javax.swing.JFrame;

public class PruebaClienteTresEnRaya
{
   public static void main( String args[] )
   {
      ClienteTresEnRaya aplicacion; // declara la aplicación cliente

      // si no hay argumentos de línea de comandos
      if ( args.length == 0 )
         aplicacion = new ClienteTresEnRaya( "127.0.0.1" ); // localhost
      else
         aplicacion = new ClienteTresEnRaya( args[ 0 ] ); // usa args

```

Figura 24.16 | Clase de prueba para el cliente de Tres en raya. (Parte 1 de 2).

```
        aplicacion.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
    } // fin de main
} // fin de la clase PruebaClienteTresEnRaya
```

Figura 24.16 | Clase de prueba para el cliente de Tres en raya. (Parte 2 de 2).

Si el mensaje que se recibe es `"Movimiento válido."`, en las líneas 134 y 135 se muestra el mensaje `"Movimiento válido, por favor espere."` y se hace una llamada al método `establecerMarca` (líneas 173 a 184) para establecer la marca del cliente en el cuadro actual (el cuadro en el que el usuario hizo clic), utilizando el método `invokeLater` de `SwingUtilities` para asegurar que las actualizaciones de la GUI ocurran en el subproceso despachador de eventos. Si el mensaje recibido es `"Movimiento inválido, intente de nuevo."` en la línea 139 se muestra el mensaje para que el usuario pueda hacer clic en un cuadro distinto. Si el mensaje recibido es `"El oponente realizo movimiento"`, en la línea 145 se lee un entero del servidor indicando a qué lugar se movió el oponente, y en las líneas 149 y 150 se coloca una marca en ese cuadro del tablero (de nuevo utilizando al método `invokeLater` de `SwingUtilities`, para asegurar que las actualizaciones en la GUI ocurran en el subproceso despachador de eventos). Si se recibe cualquier otro mensaje, en la línea 155 simplemente se muestra ese mensaje en pantalla. En la figura 24.17 se muestran capturas de pantalla de dos aplicaciones interactuando mediante el `ServidorTresEnRaya` como ejemplo.

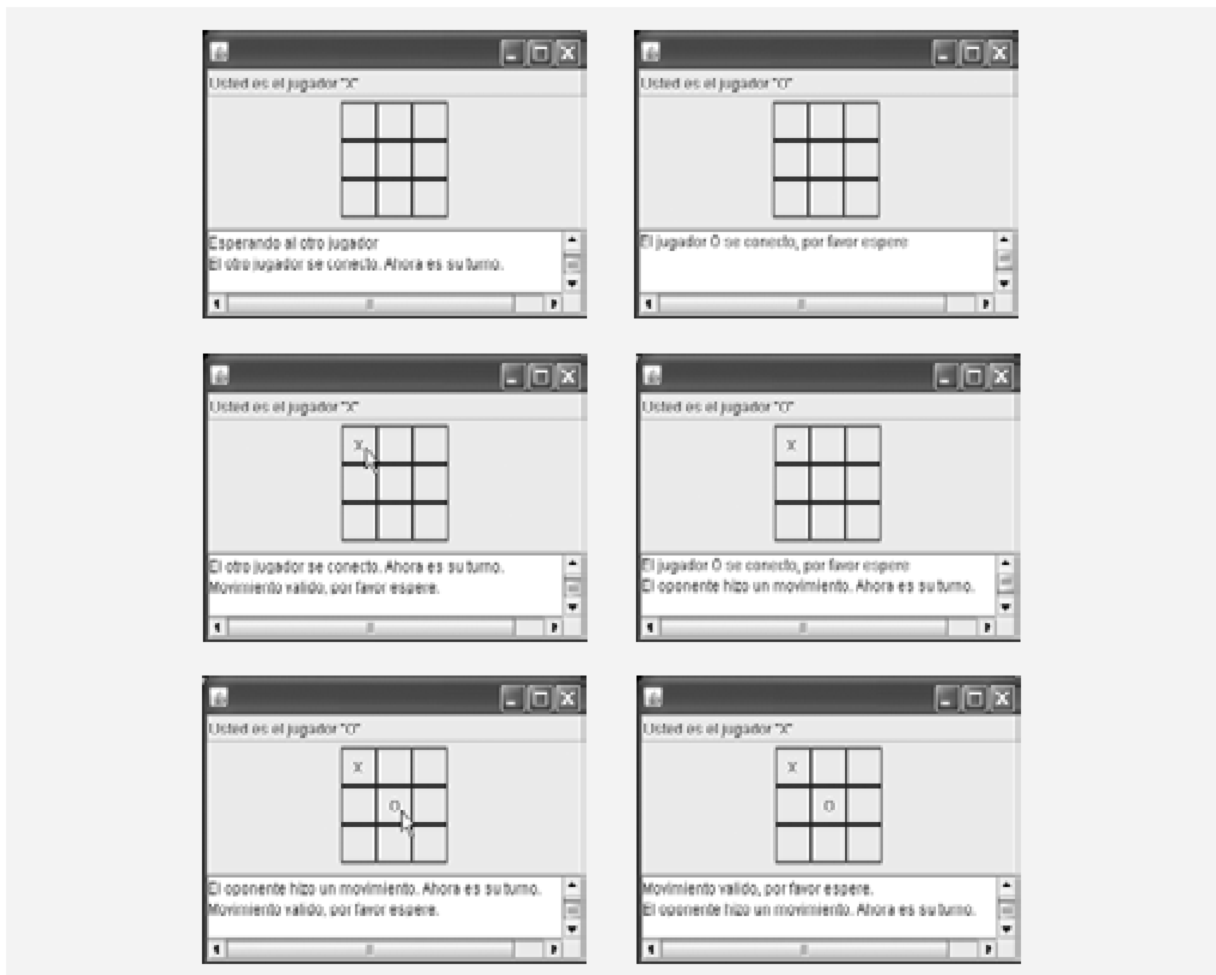


Figura 24.17 | Pantallas de salida de ejemplo del programa Tres en raya cliente/servidor. (Parte 1 de 2).

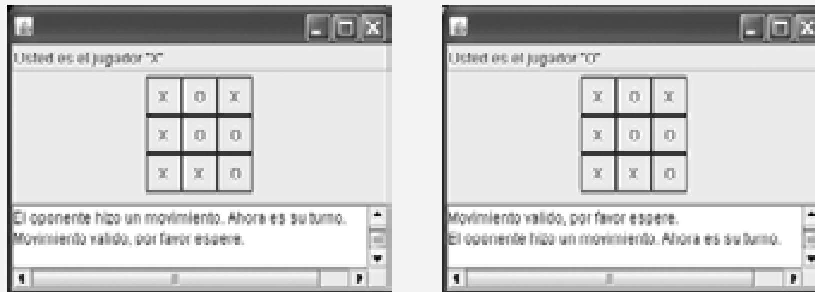


Figura 24.17 | Pantallas de salida de ejemplo del programa Tres en raya cliente/servidor. (Parte 2 de 2).

24.9 La seguridad y la red

A pesar de que sería muy interesante escribir una gran variedad de poderosas aplicaciones basadas en red, nuestros esfuerzos podrían estar limitados debido a cuestiones relacionadas con la seguridad. Muchos navegadores Web como Mozilla y Microsoft Internet Explorer prohíben, de manera predeterminada, que los applets de Java realicen un procesamiento de los archivos en los equipos en los que se ejecutan. Piense en ello. Un applet de Java está diseñado para ser enviado a su navegador a través de un documento HTML que puede descargarse de cualquier servidor Web en el mundo. Por lo general, es muy poco lo que podrá saber acerca de los orígenes de los applets de Java que se ejecuten en su sistema. Podría ser desastroso permitir que estos applets tuvieran la libertad de manipular sus archivos.

Una situación más sutil ocurre al limitar los equipos con los que los applets en ejecución pueden realizar conexiones de red. Para crear aplicaciones con una verdadera capacidad de colaboración, sería ideal que nuestros applets pudieran comunicarse con equipos en casi cualquier parte. Por lo general, el administrador de seguridad de Java en un navegador Web restringe a un applet de manera que sólo pueda comunicarse con el equipo desde el que se descargó originalmente.

Estas restricciones podrían parecer demasiado estrictas. Sin embargo, la API de seguridad de Java ahora proporciona herramientas para applets con firma digital, con lo cual los navegadores podrán determinar si un applet se descarga desde un **origen de confianza** (**trusted source**). Un applet de confianza puede recibir acceso adicional al equipo en el que se está ejecutando. Las características de la API de seguridad de Java y ciertas herramientas de red adicionales se describen en nuestro texto *Advanced Java 2 Platform How to Program*.

24.10 [Bono Web] Ejemplo práctico: servidor y cliente DeitelMessenger

[*Nota:* este ejemplo práctico está disponible en `www.deitel.com/books/jhtp7/`]. Los salones de conversación se han vuelto bastante comunes en Internet. Estos salones proporcionan una ubicación central en donde los usuarios pueden conversar entre sí, mediante mensajes de texto cortos. Cada participante en un salón de conversación puede ver todos los mensajes que publican los demás usuarios, y cada uno de los usuarios puede publicar mensajes. Este ejemplo práctico integra muchas de las características de Java relacionadas con redes, subprocesamiento múltiple y la GUI de Swing que hemos aprendido hasta ahora, para crear un sistema de conversación en línea. También presentaremos la **transmisión múltiple** (**multicasting**), que permite a una aplicación enviar objetos `DatagramPacket` a grupos de clientes.

El ejemplo práctico DeitelMessenger es una aplicación considerable que utiliza muchas características intermedias de Java, como el trabajo en red con objetos `Socket`, `DatagramPacket` y `MulticastSocket`, subprocesamiento múltiple y la GUI de Swing. En este ejemplo práctico también se demuestran las buenas prácticas de ingeniería de software al separar la interfaz de la implementación, lo cual permite a los desarrolladores dar soporte a distintos protocolos de red, y proporcionar distintas interfaces de usuario. Después de leer este ejemplo práctico, usted podrá construir aplicaciones de red más importantes.

24.11 Conclusión

En este capítulo aprendió acerca de los fundamentos de la programación de redes en Java. Conoció dos métodos distintos para enviar datos a través de una red: el trabajo en red basado en flujos mediante el uso de TCP/IP, y el trabajo en red basado en datagramas mediante el uso de UDP. En el siguiente capítulo, aprenderá acerca de los conceptos básicos de las bases de datos, a interactuar con los datos en una base de datos mediante el uso de SQL, y a utilizar JDBC para permitir que las aplicaciones de Java manipulen los datos de una base de datos.

Resumen

Sección 24.1 Introducción

- Java cuenta con sockets de flujo y sockets de datagrama. Con los sockets de flujo, un proceso establece una conexión con otro proceso. Mientras la conexión esté presente, los datos fluyen entre los procesos en flujos. Se dice que los sockets de flujo proporcionan un servicio orientado a la conexión. El protocolo que se utiliza para la transmisión es el popular TCP (Protocolo de control de transmisión).
- Con los sockets de datagrama, se transmiten paquetes individuales de información. UDP (Protocolo de datagramas de usuario) es un servicio sin conexión que no garantiza que los paquetes no se pierdan, se dupliquen o lleguen fuera de secuencia. Se requiere un esfuerzo de programación adicional de parte del programador para lidiar con estos problemas.

Sección 24.2 Manipulación de URLs

- El protocolo HTTP (Protocolo de transferencia de hipertexto) que forma la base del servicio Web utiliza URIs (Identificadores uniformes de recursos) para localizar datos en Internet. Los URIs comunes representan archivos o directorios, y pueden representar tareas complejas como búsquedas en bases de datos y en Internet. Un URI que representa a un documento se conoce como URL (Identificador uniforme de recursos).
- El método `getAppletContext` de `Applet` devuelve una referencia a un objeto `AppletContext`, el cual representa el entorno del applet (es decir, el navegador en el que se ejecuta el applet). El método `showDocument` de `AppletContext` recibe un `URL` como argumento y lo pasa al objeto `AppletContext` (es decir, el navegador), el cual muestra el recurso Web asociado con ese `URL`. Una segunda versión de `showDocument` permite a un applet especificar el marco de destino en el que se va a visualizar un recurso Web. Los marcos de destino especiales son: `_blank` (se muestra en una nueva ventana del navegador Web), `_self` (se muestra en el mismo marco que el applet) y `_top` (se eliminan los marcos actuales y después se muestra en la ventana actual).

Sección 24.3 Cómo leer un archivo en un servidor Web

- El método `setPage` de `JEditorPane` descarga el documento especificado por su argumento y lo muestra en el objeto `JEditorPane`.
- Por lo general, un documento HTML contiene hipervínculos (texto, imágenes o componentes de la GUI que, cuando se hace clic en ellos, proporcionan un acceso rápido a otro documento en Web. Si un documento HTML se muestra en un objeto `JEditorPane` y el usuario hace clic en un hipervínculo, el objeto `JEditorPane` genera un evento `HyperlinkEvent` y notifica a todos los objetos `HyperlinkListener` acerca de ese evento.
- El método `getEventType` de `HyperlinkEvent` determina el tipo del evento. `HyperlinkEvent` contiene la clase anidada `EventType`, la cual declara tres tipos de eventos de hipervínculo: `ACTIVATED` (se hizo clic en el hipervínculo), `ENTERED` (se desplazó el ratón sobre un hipervínculo) y `EXITED` (se retiró el ratón de un hipervínculo). El método `getURL` de `HyperlinkEvent` obtiene el `URL` representado por el hipervínculo.

Sección 24.4 Cómo establecer un servidor simple utilizando sockets de flujo

- Las conexiones basadas en flujo se administran con objetos `Socket`.
- Un objeto `ServerSocket` establece el puerto en el que el servidor espera las conexiones de los clientes. El segundo argumento para el constructor de `ServerSocket` es el número de conexiones que pueden esperar en una cola para conectarse con el servidor. Si la cola de clientes está llena, se rechazan las conexiones de los clientes. El método `accept` de `ServerSocket` espera de manera indefinida (es decir, se bloquea) una conexión de un cliente y devuelve un objeto `Socket` cuando se establece una conexión.
- Los métodos `getOutputStream` y `getInputStream` de `Socket` obtienen referencias a objetos `OutputStream` e `InputStream` de un objeto `Socket`, respectivamente. El método `close` de `Socket` termina una conexión.

Sección 24.5 Cómo establecer un cliente simple utilizando sockets de flujo

- Al crear un objeto Socket se especifica un nombre de servidor y un número de puerto, para que éste pueda conectar a un cliente con el servidor. Un intento de conexión fallido lanza una excepción IOException.
- El método getByName de InetAddress devuelve un objeto InetAddress que contiene el nombre de host de la computadora para la cual se especifica el nombre de host o dirección IP como argumento. El método getLocalHost de InetAddress devuelve un objeto InetAddress que contiene el nombre de host del equipo local que ejecuta el programa.

Sección 24.7 Interacción entre cliente/servidor sin conexión mediante datagramas

- La transmisión orientada a la conexión es como el sistema telefónico: usted marca y recibe una conexión al teléfono de la persona con la que usted desea comunicarse. La conexión se mantiene todo el tiempo que dure su llamada telefónica, incluso aunque usted no esté hablando.
- La transmisión sin conexión mediante datagramas es similar a la manera en que el correo se transporta mediante el servicio postal. Si un mensaje extenso no cabe en un sobre, se puede dividir en varias piezas separadas que se colocan en sobres separados, numerados en forma secuencial. Cada una de las cartas se envía entonces por correo al mismo tiempo. Las cartas podrían llegar en orden, sin orden o tal vez no llegarían.
- Los objetos DatagramPacket almacenan paquetes de datos que una aplicación va a enviar o recibir. Los objetos DatagramSocket envían y reciben objetos DatagramPacket.
- El constructor de DatagramSocket que no recibe argumentos enlaza la aplicación a un puerto elegido por la computadora en la que se ejecuta el programa. El constructor de DatagramSocket que recibe un número de puerto entero enlaza la aplicación al puerto especificado. Si un constructor de DatagramSocket no enlaza la aplicación a un puerto, se produce una excepción SocketException. El método receive de DatagramSocket se bloquea (espera) hasta que llega un paquete, y después almacena este paquete en su argumento.
- El método getAddress de DatagramPacket devuelve un objeto InetAddress que contiene información acerca del equipo host desde el que se envió el paquete. El método getPort devuelve un entero que especifica el número de puerto a través del cual el equipo host envió el objeto DatagramPacket. El método getLength devuelve un entero que representa el número de bytes de datos en un objeto DatagramPacket. El método getData devuelve un arreglo de bytes que contiene los datos en un objeto DatagramPacket.
- El constructor de DatagramPacket para un paquete que se va a enviar recibe cuatro argumentos: el arreglo de bytes que se va a enviar, el número de bytes a enviar, la dirección cliente a la que se enviará el paquete y el número de puerto en el que espera el cliente para recibir paquetes.
- El método send de DatagramSocket envía un objeto DatagramPacket a través de la red.
- Si ocurre un error al recibir o enviar un objeto DatagramPacket, se produce una excepción IOException.

Sección 24.9 La seguridad y la red

- Por lo general, los navegadores Web restringen a un applet, de manera que sólo pueda comunicarse con el equipo desde el que se descargó originalmente.

Terminología

_blank, marco de destino
_self, marco de destino
_top, marco de destino
accept, método de la clase ServerSocket
ACTIVATED, constante de la clase anidada EventType
AppletContext, interfaz
bloquear para escuchar en espera de una conexión
cargador de clases
cliente
close, método de la clase Socket
comunicación basada en sockets
comunicaciones basadas en paquetes
conexión
Consorcio World Wide Web (W3C)
DatagramPacket, clase
DatagramSocket, clase
dirección de bucle local (loopback)
encabezado de flujo
enlazar el servidor al puerto
ENTERED, constante de la clase anidada EventType
EventType, clase anidada de HyperlinkEvent
EXITED, constante de la clase anidada EventType
getAddress, método de la clase DatagramPacket
getAppletContext, método de la clase Applet
getByName, método de la clase InetAddress
getData, método de la clase DatagramPacket
getEventType, método de la clase HyperlinkEvent
getHostName, método de la clase InetAddress
getInetAddress, método de la clase Socket
getInputStream, método de la clase Socket
getLength, método de la clase DatagramPacket
getLocalHost, método de la clase InetAddress
getOutputStream, método de la clase Socket
getParameter, método de la clase Applet

getPort, método de la clase DatagramPacket
getResource, método de la clase Class
getURL, método de la clase HyperlinkEvent
HyperlinkEvent, clase
HyperlinkListener, interfaz
hyperlinkUpdate, método de la interfaz HyperlinkListener
Identificador uniforme de recursos (URI)
InetAddress, clase
java.net, paquete
JEditorPane, clase
longitud de cola
MalformedURLException, clase
marco de destino
marco de HTML
número de puerto
origen de confianza
paquete
paquete de datagrama
param, etiqueta
parámetro de applet
Protocolo de datagramas de usuario (UDP)
Protocolo de transferencia de hipertexto (HTTP)
puerto
punto de negociación (handshake)
receive, método de la clase DatagramSocket
registrar un puerto
relación cliente/servidor
repite un paquete y lo envía de vuelta al cliente
send, método de la clase DatagramSocket
ServerSocket, clase
servicio orientado a la conexión
servicio sin conexión
servidor
setPage, método de la clase JEditorPane
showDocument, método de la clase AppletContext
socket
socket de datagrama
socket de flujo
Socket, clase
SocketException, clase
TCP (Protocolo de control de transmisión)
transmisión basada en flujos, orientada a la conexión
transmisión basada en sockets
transmisión sin conexión
UDP (Protocolo de datagramas de usuario)
UnknownHostException, clase

Ejercicios de autoevaluación

24.1 Complete las siguientes oraciones:

a) La excepción ________________ ocurre cuando se produce un error de entrada/salida al cerrar un socket.
b) La excepción ________________ ocurre cuando un nombre de host indicado por un cliente no se puede resolver a una dirección.
c) Si un constructor de DatagramSocket no puede establecer un objeto DatagramSocket apropiadamente, se produce una excepción del tipo ________________.
d) Muchas de las clases para trabajo en red de Java están contenidas en el paquete ________________.
e) La clase ________________ enlaza la aplicación a un puerto para la transmisión de datagramas.
f) Un objeto de la clase ________________ contiene una dirección IP.
g) Los dos tipos de sockets que vimos en este capítulo son ________________ y ________________.
h) El acrónimo URL significa ________________.
i) El acrónimo URI significa ________________.
j) El protocolo clave que forma la base de la World Wide Web es ________________.
k) El método ________________ de AppletContext recibe un objeto URL como argumento y muestra en un navegador el recurso Web asociado con ese URL.
l) El método getLocalHost devuelve un objeto ________________ que contiene el nombre de host local de la computadora en la que se está ejecutando el programa.
m) El constructor de URL determina si su argumento de cadena es un URL válido. De ser así, el objeto URL se inicializa con esa ubicación; en caso contrario, se produce una excepción ________________.

24.2 Conteste con *verdadero* o *falso* a cada una de las siguientes proposiciones; en caso de ser *falso*, explique por qué

a) UDP es un protocolo orientado a la conexión.
b) Con los sockets de flujo, un proceso establece una conexión con otro proceso.
c) Un servidor espera en un puerto las conexiones de un cliente.
d) La transmisión de paquetes con datagramas a través de una red es un proceso confiable; se garantiza que los paquetes lleguen en secuencia.

e) Por razones de seguridad, muchos navegadores Web como Mozilla permiten a los applet de Java realizar el procesamiento de archivos solamente en los equipos en los que se ejecutan.
f) A menudo, los navegadores Web restringen a un applet de manera que sólo pueda comunicarse con el equipo desde el que se descargó originalmente.

Respuestas a los ejercicios de autoevaluación

24.1 a) `IOException`. b) `UnknownHostException`. c) `SocketException`. d) `java.net`. e) `DatagramSocket`. f) `InetAddress`. g) sockets de flujo, sockets de datagrama. h) Localizador uniforme de recursos. i) Identificador uniforme de recursos. j) http. k) `showDocument`. l) `InetAddress`. m) `MalformedURLException`.

24.2 a) Falso; UDP es un protocolo sin conexión y TCP es un protocolo orientado a la conexión. b) Verdadero. c) Verdadero. d) Falso; los paquetes podrían perderse, llegar desordenados o duplicarse. e) Falso; la mayoría de los navegadores evita que los applets realicen un procesamiento de archivos en el equipo cliente. f) Verdadero.

Ejercicios

24.3 Indique la diferencia entre los servicios de red orientados a la conexión y los servicios sin conexión.

24.4 ¿Cómo determina un cliente el nombre de host del equipo cliente?

24.5 ¿Bajo qué circunstancias se lanzaría una excepción `SocketException`?

24.6 ¿Cómo puede un cliente obtener una línea de texto de un servidor?

24.7 Describa cómo se conecta un cliente con un servidor.

24.8 Describa cómo un servidor envía datos a un cliente.

24.9 Describa cómo operar un servidor para recibir una petición de conexión basada en flujo de un solo cliente.

24.10 ¿Cómo escucha un servidor las conexiones de sockets basadas en flujo en un puerto?

24.11 ¿Qué es lo que determina cuántas peticiones de conexión de los clientes pueden esperar en una cola para conectarse con un servidor?

24.12 Según lo descrito en el texto, ¿cuáles son las razones que podrían ocasionar que un servidor rechazara una petición de conexión de un cliente?

24.13 Use una conexión de socket para permitir a un cliente especificar un nombre de archivo, y haga que el servidor envíe el contenido del mismo o indique que el archivo no existe.

24.14 Modifique el ejercicio 24.13 para permitir al cliente modificar el contenido del archivo y enviarlo de vuelta al servidor para que lo almacene. El usuario puede editar el archivo en un objeto `JTextArea` y después hacer clic en un botón *guardar cambios* para enviar el archivo de vuelta al servidor.

24.15 Modifique el programa de la figura 24.2 para permitir que los usuarios agreguen sus propios sitios a la lista, y que también los eliminen de la lista.

24.16 Los servidores con subprocesamiento múltiple son bastante populares hoy en día, especialmente debido al uso cada vez mayor de servidores con multiprocesamiento. Modifique la aplicación de servidor simple presentada en la sección 24.6, para que sea un servidor con subprocesamiento múltiple. Después utilice varias aplicaciones cliente y haga que cada una de ellas se conecte al servidor en forma simultánea. Use un objeto `ArrayList` para almacenar los subprocesos cliente. `ArrayList` proporciona varios métodos que pueden utilizarse en este ejercicio. El método `size` determina el número de elementos en un objeto `ArrayList`. El método `get` devuelve el elemento en la ubicación especificada por su argumento. El método `add` coloca su argumento al final del objeto `ArrayList`. El método `remove` elimina su argumento del objeto `ArrayList`.

24.17 *(Juego de damas)* En el texto presentamos un programa de Tres en raya, controlado por un servidor con subprocesamiento múltiple. Desarrolle un programa de damas que se modele en base al programa de Tres en raya. Los dos usuarios deberán alternar sus movimientos. Su programa debe mediar los movimientos de los jugadores, determinando el turno de cada quién y permitiendo sólo movimientos válidos. Los mismos jugadores determinarán cuando el juego se haya acabado.

24.18 *(Juego de ajedrez)* Desarrolle un programa para jugar ajedrez, modelado en base al programa de damas del ejercicio 24.17.

24.19 *(Juego de Blackjack)* Desarrolle un programa para jugar cartas estilo Blackjack, en el que la aplicación servidor reparta las cartas a cada uno de los applets cliente. El servidor deberá repartir cartas adicionales (según las reglas del juego) a cada jugador, según sea requerido.

24.20 *(Juego de Póquer)* Desarrolle un programa para jugar cartas estilo Póquer, en el que la aplicación servidor reparta las cartas a cada uno de los applets cliente. El servidor deberá repartir cartas adicionales (según las reglas del juego) a cada jugador, según sea requerido.

24.21 *(Modificaciones al programa de Tres en raya con subprocesamiento múltiple)* Los programas de las figuras 24.13 y 24.15 implementan una versión cliente/servidor con subprocesamiento múltiple del juego Tres en raya. Nuestro objetivo al desarrollar este juego fue demostrar el uso de un servidor con subprocesamiento múltiple que pudiera procesar varias conexiones de clientes al mismo tiempo. El servidor en el ejemplo realmente es un mediador entre los dos applets cliente; se asegura que cada movimiento sea válido y que cada cliente se mueva en el orden apropiado. El servidor no determina quién ganó o perdió, o si hubo un empate. Además, no existe la capacidad de permitir que se inicie un juego nuevo, o de terminar un juego existente.

A continuación se muestra una lista de modificaciones sugeridas a las figuras 24.13 y 24.15:

a) Modificar la clase `ServidorTresEnRaya` para probar si se ganó, perdió o empató en cada movimiento del juego. Enviar un mensaje a cada applet cliente que indique el resultado del juego cuando éste termine.

b) Modificar la clase `ClienteTresEnRaya` para mostrar un botón que, al hacer clic sobre él, permita al cliente jugar otro juego. El botón deberá habilitarse sólo cuando se complete un juego. Observe que las clases `ClienteTresEnRaya` y `ServidorTresEnRaya` deben modificarse para restablecer el tablero y toda la demás información de estado. Además, el otro `ClienteTresEnRaya` debe ser notificado cuando un nuevo juego esté por empezar, para que su tablero y su estado puedan restablecerse.

c) Modificar la clase `ClienteTresEnRaya` para mostrar un botón que, al hacer clic sobre él, el cliente pueda terminar el programa en cualquier momento. Cuando el usuario haga clic en el botón, el servidor y el otro cliente deberán ser notificados. Después el servidor deberá esperar una conexión de otro cliente, para que pueda empezar un juego nuevo.

d) Modifique las clases `ClienteTresEnRaya` y `ServidorTresEnRaya` de manera que el ganador de un juego pueda seleccionar la pieza X u O para el siguiente juego. Recuerde: la X siempre va primero.

e) Si le gusta ser ambicioso, permita a un cliente jugar contra el servidor mientras éste espera la conexión de otro cliente.

24.22 *(Tres en raya con subprocesamiento múltiple, en 3-D)* Modifique el programa de Tres en raya cliente/servidor con subprocesamiento múltiple, para implementar una versión tridimensional del juego, de $4 \times 4 \times 4$. Implemente la aplicación servidor para que sea el intermediario entre los dos clientes. Muestre el tablero tridimensional como cuatro tableros que contienen cuatro filas y cuatro columnas cada uno. Si le gusta ser ambicioso, trate de hacer las siguientes modificaciones:

a) Dibuje el tablero en forma tridimensional.

b) Permita al servidor probar si hay un ganador, un perdedor o si fue empate. ¡Cuidado! ¡Hay muchas posibles maneras de ganar en un tablero de $4 \times 4 \times 4$!

24.23 *(Código Morse en red)* Tal vez el más famoso de todos los esquemas de codificación sea el código Morse, desarrollado por Samuel Morse en 1832 para usarlo con el sistema telegráfico. El código Morse asigna una serie de puntos y guiones cortos a cada letra del alfabeto, a cada dígito y a unos cuantos caracteres especiales (punto, coma, signo de dos puntos y signo de punto y coma). En los sistemas orientados al sonido, el punto representa un sonido corto y el guión corto representa un sonido largo. En los sistemas orientados a la luz y en los sistemas de señalización con banderas se utilizan otras representaciones de puntos y guiones cortos. La separación entre las palabras se indica mediante un espacio o, simplemente, la ausencia de un punto o un guión corto. En un sistema orientado al sonido, un espacio se indica mediante un lapso corto de tiempo durante el cual no se transmite ningún sonido. La versión internacional del código Morse aparece en la figura 24.18.

Escriba una aplicación cliente/servidor en la que dos clientes puedan enviarse entre sí mensajes en código Morse, a través de una aplicación servidor con subprocesamiento múltiple. La aplicación cliente deberá permitir al usuario escribir caracteres normales en un objeto `JTextArea`. Cuando el usuario envíe el mensaje, la aplicación cliente deberá traducir los caracteres en código Morse y enviar el mensaje codificado a través del servidor, al otro cliente. Use un espacio en blanco entre cada letra en código Morse y tres espacios en blanco entre cada palabra en código Morse. Cuando se reciban los mensajes, deberán ser decodificados y mostrados como caracteres normales y como código Morse. El cliente debe tener un objeto `JTextArea` para escribir y un objeto `JTextArea` para mostrar los mensajes del otro cliente.

Carácter	Código	Carácter	Código
A	.-	T	-
B	-...	U	..-
C	-.-.	V	...-
D	-..	W	.--
E	.	X	-..-
F	..-.	Y	-.--
G	--.	Z	--..
H	....		
I	..	*Dígitos*	
J	.---	1	.----
K	-.-	2	..---
L	.-..	3	...--
M	--	4	....-
N	-.	5	.....
O	---	6	-....
P	.--.	7	--...
Q	--.-	8	---..
R	.-.	9	----.
S	...	0	-----

Figura 24.18 | Las letras del alfabeto, expresadas en código Morse internacional.

25



Es un error capital especular antes de tener datos.

—Arthur Conan Doyle

Ahora ven, escríbelo en una tablilla, grábalo en un libro, y que dure hasta el último día, para testimonio.

—La Santa Biblia, Isaías 30:8

Primero consiga los hechos, y después podrá distorsionarlos según le parezca.

—Mark Twain

Me gustan dos tipos de hombres: domésticos y foráneos.

—Mae West

Acceso a bases de datos con JDBC

OBJETIVOS

En este capítulo aprenderá a:

- Utilizar los conceptos acerca de las bases de datos relacionales.
- Utilizar el Lenguaje de Consulta Estructurado (SQL) para obtener y manipular los datos de una base de datos.
- Utilizar la API JDBC™ del paquete `java.sql` para acceder a las bases de datos.
- Utilizar la interfaz `RowSet` del paquete `javax.sql` para manipular bases de datos.
- Utilizar el descubrimiento de controladores de JDBC automático de JDBC 4.0.
- Utilizar objetos `PreparedStatement` para crear instrucciones de SQL precompiladas con parámetros.
- Conocer cómo el procesamiento de transacciones hace que las aplicaciones de datos sea más robusto.

Plan general

25.1 Introducción

Una **base de datos** es una colección organizada de datos. Existen diversas estrategias para organizar datos y facilitar el acceso y la manipulación. Un **sistema de administración de bases de datos (DBMS)** proporciona los mecanismos para almacenar, organizar, obtener y modificar datos para muchos usuarios. Los sistemas de administración de bases de datos permiten el acceso y almacenamiento de datos sin necesidad de preocuparse por su representación interna.

En la actualidad, los sistemas de bases de datos más populares son las bases de datos relacionales, en donde los datos se almacenan sin considerar su estructura física (sección 25.2). Un lenguaje llamado **SQL** es el lenguaje estándar internacional que se utiliza casi universalmente con las bases de datos relacionales para realizar **consultas** (es decir, para solicitar información que satisfaga ciertos criterios) y para manipular datos.

Algunos **sistemas de administración de bases de datos relacionales (RDBMS)** populares son Microsoft SQL Server, Oracle, Sybase, IBM DB2, Informix, PostgreSQL y MySQL. El JDK incluye ahora un RDBMS puro de Java, conocido como Java DB (la versión de Sun de Apache Derby). En este capítulo presentaremos ejemplos utilizando MySQL y Java DB.[1]

---

1. MySQL es uno de los sistemas de administración de bases de datos de código fuente abierto más populares de la actualidad. Al momento de escribir este libro todavía no soportaba JDBC 4, que forma parte de Java SE 6 (Mustang). Sin embargo, el sistema Java DB de Sun, que está basado en el sistema de administración de bases de datos de código fuente abierto Apache Derby y se incluye con el JDK 1.6.0 de Sun, sí ofrece soporte para JDSBC 4. En las secciones 25.8 a 25.10 utilizamos MySQL y JDBC 3, y en la sección 25.11 utilizamos Java DB y JDBC 4.

Los programas en Java se comunican con las bases de datos y manipulan sus datos utilizando la **API JDBC™**. Un **controlador de JDBC** permite a las aplicaciones de Java conectarse a una base de datos en un DBMS específico, y nos permite manipular esa base de datos mediante la API JDBC.



Observación de ingeniería de software 25.1

Al utilizar la API JDBC, los desarrolladores pueden modificar el DBMS subyacente sin necesidad de modificar el código de Java que accede a la base de datos.

La mayoría de los sistemas de administración de bases de datos populares incluyen ahora controladores de JDBC. También hay muchos controladores de JDBC de terceros disponibles. En este capítulo presentaremos la tecnología JDBC y la emplearemos para manipular bases de datos de MySQL y Java DB. Las técnicas que demostraremos aquí también pueden usarse para manipular otras bases de datos que tengan controladores de JDBC. Consulte la documentación de su DBMS para determinar si incluye un controlador de JDBC. Incluso si su DBMS no viene con un controlador de JDBC, muchos distribuidores independientes proporcionan estos controladores para una amplia variedad de sistemas DBMS.

Para obtener más información acerca de JDBC, visite la página:

```
java.sun.com/javase/technologies/database/index.jsp
```

Este sitio contiene información relacionada con JDBC, incluyendo las especificaciones de JDBC, preguntas frecuentes (FAQs) acerca de JDBC, un centro de recursos de aprendizaje y descargas de software para buscar controladores de JDBC para su DBMS,

```
developers.sun.com/product/jdbc/drivers/
```

Este sitio proporciona un motor de búsqueda para ayudarlo a localizar los controladores apropiados para su DBMS.

25.2 Bases de datos relacionales

Una **base de datos relacional** es una representación lógica de datos que permite acceder a éstos sin necesidad de considerar su estructura física. Una base de datos relacional almacena los datos en **tablas**. En la figura 25.1 se muestra una tabla de ejemplo que podría utilizarse en un sistema de personal. El nombre de la tabla es `Empleado`, y su principal propósito es almacenar los atributos de un empleado. Las tablas están compuestas de **filas**, y las filas, de **columnas** en las que se almacenan los valores. Esta tabla consiste de seis filas. La columna `Numero` de cada fila en esta tabla es su **clave primaria**: una columna (o grupo de columnas) en una tabla que tiene un valor único, el cual no puede duplicarse en las demás filas. Esto garantiza que cada fila pueda identificarse por su clave primaria. Algunos buenos ejemplos de columnas con clave primaria son un número de seguro social, un número de identificación de empleado y un número de pieza en un sistema de inventario, ya que se garantiza que los valores en cada una de esas columnas serán únicos. Las filas de la figura 25.1 se muestran en orden, con base en la clave primaria. En este caso, las filas se muestran en orden ascendente; también podríamos utilizar el orden descendente.

Número	Nombre	Departamento	Salario	Ubicación
23603	Jones	413	1100	New Jersey
24568	Kerwin	413	2000	New Jersey
34589	Larson	642	1800	Los Angeles
35761	Myers	611	1400	Orlando
47132	Neumann	413	9000	New Jersey
78321	Stephens	611	8500	Orlando

Fila
Clave primaria
Columna

Figura 25.1 | Datos de ejemplo de la tabla `Empleado`.

No se garantiza que las filas en las tablas se almacenen en un orden específico. Como lo demostraremos en un ejemplo más adelante, los programas pueden especificar criterios de ordenamiento al solicitar datos de una base de datos.

Cada columna de la tabla representa un atributo de datos distinto. Las filas generalmente son únicas (por clave primaria) dentro de una tabla, pero los valores de columnas específicas pueden duplicarse entre filas. Por ejemplo, tres filas distintas en la columna `Departamento` de la tabla `Empleado` contienen el número `413`.

A menudo los distintos usuarios de una base de datos se interesan en datos diferentes, y en relaciones distintas entre esos datos. La mayoría de los usuarios requieren solamente de ciertos subconjuntos de las filas y columnas. Para obtener estos subconjuntos, utilizamos consultas para especificar cuáles datos se deben seleccionar de una tabla. Utilizamos SQL para definir consultas complejas que seleccionen datos de una tabla. Por ejemplo, podríamos seleccionar datos de la tabla `Empleado` para crear un resultado que muestre en dónde se ubican los departamentos, y presentar los datos ordenados en forma ascendente, por número de departamento. Este resultado se muestra en la figura 25.2. Hablaremos sobre las consultas de SQL en la sección 25.4.

```
Departamento     Ubicación

413              New Jersey
611              Orlando
642              Los Angeles
```

Figura 25.2 | Resultado de seleccionar distintos datos de `Departamento` y `Ubicacion` de la tabla `Empleado`.

25.3 Generalidades acerca de las bases de datos relacionales: la base de datos `libros`

Ahora veremos las generalidades sobre las bases de datos relacionales, y para ello emplearemos una base de datos llamada `libros`, misma que creamos para este capítulo. Antes de hablar sobre SQL, veremos una descripción general de las tablas de la base de datos `libros`. Utilizaremos esta base de datos para presentar varios conceptos de bases de datos, incluyendo el uso de SQL para obtener información de la base de datos y manipular los datos. Le proporcionaremos una secuencia de comandos (script) para crear la base de datos. En el directorio de ejemplos para este capítulo (en el CD que acompaña al libro) encontrará la secuencia de comandos. En la sección 25.5 le explicaremos cómo utilizar esta secuencia de comandos.

La base de datos consiste de tres tablas: `autores`, `isbnAutor` y `titulos`. La tabla `autores` (descrita en la figura 25.3) consta de tres columnas que mantienen el número único de identificación de cada autor, su nombre de pila y apellido. La figura 25.4 contiene datos de ejemplo de la tabla `autores` de la base de datos `libros`.

La tabla `isbnAutor` (descrita en la figura 25.5) consta de dos columnas que representan a cada ISBN y el correspondiente número de ID del autor. Esta tabla asocia a los autores con sus libros. Ambas columnas son claves externas que representan la relación entre las tablas `autores` y `titulos`; una fila en la tabla `autores` puede estar asociada con muchas filas en la tabla `titulos` y viceversa. La figura 25.6 contiene datos de ejemplo de la tabla `isbnAutor` de la base de datos `libros`. [*Nota:* para ahorrar espacio, dividimos el contenido de esta tabla en

Columna	Descripción
`idAutor`	El número de identificación (ID) del autor en la base de datos. En la base de datos `libros`, esta columna de enteros se define como **autoincrementada**; para cada fila insertada en esta tabla, la base de datos incrementa automáticamente el valor de `idAutor` por 1 para asegurar que cada fila tenga un `idAutor` único. Esta columna representa la clave primaria de la tabla.
`nombrePila`	El nombre de pila del autor (una cadena).
`apellidoPaterno`	El apellido paterno del autor (una cadena).

Figura 25.3 | La tabla `autores` de la base de datos `libros`.

idAutor	nombrePila	apellidoPaterno
1	Harvey	Deitel
2	Paul	Deitel
3	Tem	Nieto
4	Sean	Santry

Figura 25.4 | Datos de ejemplo de la tabla `autores`.

Columna	Descripción
`idAutor`	El número de identificación (ID) del autor, una clave externa para la tabla `autores`.
`isbn`	El ISBN para un libro, una clave externa para la tabla `titulos`.

Figura 25.5 | La tabla `isbnAutor` de la base de datos `libros`.

idAutor	isbn	idAutor	isbn
1	0131869000	2	0131450913
2	0131869000	1	0131828274
1	0131483986	2	0131828274
2	0131483986	3	0131450913
1	0131450913	4	0131828274

Figura 25.6 | Datos de ejemplo de la tabla `isbnAutor` de `libros`.

dos columnas, cada una de las cuales contiene las columnas `idAutor` y `isbn`.] La columna `idAutor` es una **clave externa**; una columna en esta tabla que coincide con la columna de la clave primaria en otra tabla (por ejemplo, `idAutor` en la tabla `autores`). Las claves externas se especifican al crear una tabla. La clave externa ayuda a mantener la **Regla de integridad referencial**: todo valor de una clave externa debe aparecer como el valor de la clave primaria de otra tabla. Esto permite al DBMS determinar si el valor de `idAutor` para un libro específico es válido. Las claves externas también permiten seleccionar datos relacionados en varias tablas para fines analíticos; a esto se conoce como **unir** los datos.

La tabla `titulos` descrita en la figura 25.7 consiste en cuatro columnas que representan el ISBN, el título, el número de edición y el año de copyright. La tabla está en la figura 25.8.

Hay una relación de uno a varios entre una clave primaria y su correspondiente clave externa (por ejemplo, una editorial puede publicar muchos libros). Una clave externa puede aparecer varias veces en su propia tabla, pero sólo una vez (como clave primaria) en otra tabla. La figura 25.9 es un **diagrama de relación de entidades (ER)** para la base de datos `libros`. Este diagrama muestra las tablas en la base de datos, así como las relaciones entre ellas. El primer compartimiento en cada cuadro contiene el nombre de la tabla. Los nombres en cursiva son claves primarias. La clave primaria de una tabla identifica en forma única a cada fila. Cada fila debe tener un valor en la clave primaria, y éste debe ser único en la tabla. A esto se le conoce como **Regla de integridad de entidades**.



Error común de programación 25.1

Si no se proporciona un valor para cada columna en una clave primaria, se quebranta la Regla de Integridad de Entidades y el DBMS reporta un error.

Columna	Descripción
`isbn`	El número ISBN del libro (una cadena). La clave primaria de la tabla. ISBN son las siglas de "International Standard Book Number" (Número internacional normalizado para libros); un esquema de numeración utilizado por las editoriales en todo el mundo para dar a cada libro un número de identificación único.
`titulo`	Título del libro (una cadena).
`numeroEdicion`	Número de edición del libro (un entero).
`copyright`	Año de edición (copyright) del libro (una cadena).

Figura 25.7 | La tabla `titulos` de la base de datos `libros`.

isbn	titulo	numeroEdicion	copyright
0131869000	Visual Basic How to Program	3	2006
0131525239	Visual C# How to Program	2	2006
0132222205	Java How to Program	7	2007
0131857576	C++ How to Program	5	2005
0132404168	C How to Program	5	2007
0131450913	Internet & World Wide Web How to Program	3	2004

Figura 25.8 | Datos de ejemplo para la tabla `titulos` de la base de datos `libros`.

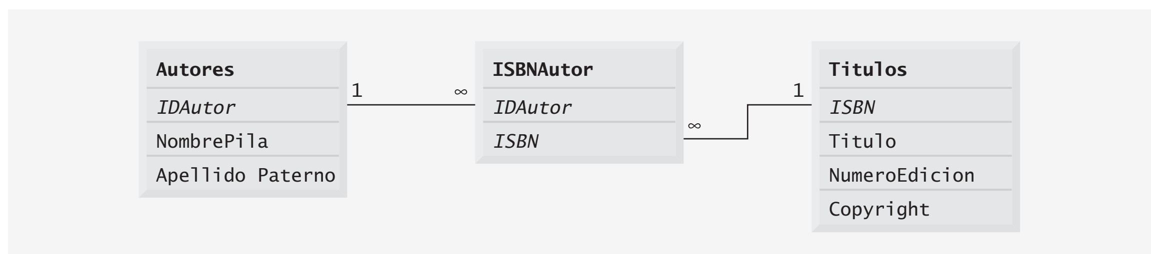


Figura 25.9 | Relaciones de las tablas en la base de datos `libros`.



Error común de programación 25.2

Al proporcionar el mismo valor para la clave primaria en varias filas, el DBMS reporta un error.

Las líneas que conectan las tablas en la figura 25.9 representan las relaciones entre las tablas. Considere la línea entre las tablas `isbnAutor` y `autores`. En el extremo de la línea que va a `autores` hay un `1`, y en el extremo que va a `isbnAutor` hay un símbolo de infinito (∞), el cual indica una **relación de uno a varios** en la que cualquier autor de la tabla `autores` puede tener un número arbitrario de libros en la tabla `isbnAutor`. Observe que la línea de relación enlaza a la columna `idAutor` en la tabla `autores` (su clave primaria) con la columna `idAutor` en la tabla `isbnAutor` (es decir, su clave externa). La columna `idAutor` en la tabla `isbnAutor` es una clave externa.



Error común de programación 25.3

Al proporcionar un valor de clave externa que no aparezca como valor de clave primaria en otra tabla, se quebranta la Regla de Integridad Referencial y el DBMS reporta un error.

La línea entre las tablas `titulos` e `isbnAutor` muestra otra relación de uno a varios; un título puede ser escrito por cualquier número de autores. De hecho, el único propósito de la tabla `isbnAutor` es proporcionar una relación de varios a varios entre las tablas `autores` y `titulos`; un autor puede escribir cualquier número de libros y un libro puede tener cualquier número de autores.

25.4 SQL

En esta sección veremos una descripción general acerca de SQL, en el contexto de nuestra base de datos `libros`. En los ejemplos posteriores y en los ejemplos de los capítulos 26 a 28, usted podrá utilizar las consultas de SQL que describimos aquí.

Las palabras clave de SQL enlistadas en la figura 25.10 se describen en las siguientes subsecciones, dentro del contexto de consultas e instrucciones de SQL completas. Las demás palabras clave de SQL se encuentran más allá del alcance de este libro. Para aprender acerca de las otras palabras clave, consulte la guía de referencia de SQL que proporciona el distribuidor del RDBMS que usted utilice. [*Nota:* para obtener más información acerca de SQL, consulte los recursos Web en la sección 25.15 y las lecturas recomendadas que se enlistan al final de este capítulo].

Palabra clave de SQL	Descripción
`SELECT`	Obtiene datos de una o más tablas.
`FROM`	Las tablas involucradas en la consulta. Se requiere para cada `SELECT`.
`WERE`	Los criterios de selección que determinan cuáles filas se van a recuperar, eliminar o actualizar. Es opcional en una consulta o instrucción de SQL.
`GROUP BY`	Criterios para agrupar filas. Es opcional en una consulta `SELECT`.
`ORDER BY`	Criterios para ordenar filas. Es opcional en una consulta `SELECT`.
`INNER JOIN`	Fusionar filas de varias tablas.
`INSERT`	Insertar filas en una tabla especificada.
`UPDATE`	Actualizar filas en una tabla especificada.
`DELETE`	Eliminar filas de una tabla especificada.

Figura 25.10 | Palabras clave para consultas de SQL.

25.4.1 Consulta básica SELECT

Veamos ahora varias consultas de SQL que extraen información de la base de datos `libros`. Una consulta de SQL "selecciona" filas y columnas de una o más tablas en una base de datos. Dichas selecciones se llevan a cabo mediante consultas con la palabra clave `SELECT`. La forma básica de una consulta `SELECT` es:

`SELECT * FROM` *nombreDeTabla*

en la consulta anterior, el **asterisco (*)** indica que deben obtenerse todas las columnas de la tabla *nombreDeTabla*. Por ejemplo, para obtener todos los datos en la tabla `autores`, podemos utilizar la siguiente consulta:

```
SELECT * FROM autores
```

La mayoría de los programas no requieren todos los datos en la tabla. Para obtener sólo ciertas columnas de una tabla, reemplace el asterisco (*) con una lista separada por comas de los nombres de las columnas. Por ejemplo, para obtener solamente las columnas `idAutor` y `apellidoPaterno` para todas las filas en la tabla `autores`, utilice la siguiente consulta:

```
SELECT idAutor, apellidoPaterno FROM autores
```

Esta consulta devuelve los datos que se muestran en la figura 25.11.

idAutor	apellidoPaterno
1	Deitel
2	Deitel
3	Nieto
4	Santry

Figura 25.11 | Datos de ejemplo para `idAutor` y `apellidoPaterno` de la tabla `autores`.



Observación de ingeniería de software 25.2

Para la mayoría de las consultas, no debe utilizarse el asterisco (*) para especificar nombres de columnas. En general, los programadores procesan los resultados sabiendo de antemano el orden de las columnas en el resultado; por ejemplo, al seleccionar `idAutor` *y* `apellidoPaterno` *de la tabla* `autores` *nos aseguramos que las columnas aparezcan en el resultado con* `idAutor` *como la primera columna y* `apellidoPaterno` *como la segunda. Generalmente los programas procesan las columnas de resultado especificando el número de columna en el resultado (empezando con el número 1 para la primera columna). Al seleccionar las columnas por nombre, también evitamos devolver columnas innecesarias y nos protegemos contra los cambios en el orden actual de las columnas en la(s) tabla(s).*



Error común de programación 25.4

Si un programador supone que las columnas siempre se devuelven en el mismo orden de una consulta que utiliza el asterisco (*), el programa podría procesar los resultados incorrectamente. Si cambia el orden de las columnas en la(s) tabla(s), o si se agregan más columnas posteriormente, el orden de las columnas en el resultado cambia de manera acorde.

25.4.2 La cláusula WHERE

En la mayoría de los casos es necesario localizar, en una base de datos, filas que cumplan con ciertos **criterios de selección**. Sólo se seleccionan las filas que cumplan con los criterios de selección (formalmente llamados **predicados**). SQL utiliza la **cláusula** `WHERE` opcional en una consulta para especificar los criterios de selección para la misma. La forma básica de una consulta con criterios de selección es:

SELECT *nombreDeColumna1*, *nombreDeColumna2*, … FROM *nombreDeTabla* WHERE *criterios*

Por ejemplo, para seleccionar las columnas `titulo`, `numeroEdicion` y `copyright` de la tabla `titulos` para las cuales la fecha de `copyright` sea mayor que `2005`, utilice la siguiente consulta:

```
SELECT titulo, numeroEdicion, copyright
    FROM titulos
    WHERE copyright > '2005'
```

En la figura 25.12 se muestra el resultado de la consulta anterior. Los criterios de la cláusula `WHERE` pueden contener los operadores <, >, <=, >=, =, <> y `LIKE`. El operador `LIKE` se utiliza para **relacionar patrones** con los caracteres comodines **porcentaje (%)** y **guión bajo (_)**. El relacionar patrones permite a SQL buscar cadenas que coincidan con un patrón dado.

Un patrón que contenga un carácter de porcentaje (%) busca cadenas que tengan cero o más caracteres en la posición del carácter de porcentaje en el patrón. Por ejemplo, la siguiente consulta localiza las filas de todos los autores cuyo apellido paterno empiece con la letra `D`:

```
SELECT idAutor, nombrePila, apellidoPaterno
    FROM autores
    WHERE apellidoPaterno LIKE 'D%'
```

La consulta anterior selecciona las dos filas que se muestran en la figura 25.13, ya que dos de los cuatro autores en nuestra base de datos tienen un apellido paterno que empieza con la letra `D` (seguida de cero o más caracteres). El signo de `%` y el operador `LIKE` de la cláusula `WHERE` indican que puede aparecer cualquier número de caracteres después de la letra `D` en la columna `apellidoPaterno`. Observe que la cadena del patrón está encerrada entre caracteres de comilla sencilla.

titulo	numeroEdicion	copyright
Visual C# How to Program	2	2006
Visual Basic 2005 How to Program	3	2006
Java How to Program	7	2007
C How to Program	5	2005

Figura 25.12 | Ejemplos de títulos con fechas de copyright posteriores a 2005 de la tabla `titulos`.

idAutor	nombrePila	apellidoPaterno
1	Harvey	Deitel
2	Paul	Deitel

Figura 25.13 | Autores, cuyo apellido paterno empieza con `D` de la tabla `autores`.



Tip de portabilidad 25.1

Consulte la documentación de su sistema de bases de datos para determinar si SQL es susceptible al uso de mayúsculas en su sistema, y para determinar la sintaxis de las palabras clave de SQL (por ejemplo, ¿deben estar completamente en mayúsculas, completamente en minúsculas o puede ser una combinación de ambas?).



Tip de portabilidad 25.2

Lea cuidadosamente la documentación de su sistema de bases de datos para determinar si éste soporta el operador `LIKE`. El SQL que describimos es soportado por la mayoría de los RDBMS, pero siempre es buena idea comprobar las características de SQL que soporta su RDBMS.

Un guión bajo (_) en la cadena del patrón indica un carácter comodín individual en esa posición. Por ejemplo, la siguiente consulta localiza las filas de todos los autores cuyo apellido paterno empiece con cualquier carácter (lo que se especifica con _), seguido por la letra `o`, seguida por cualquier número de caracteres adicionales (lo que se especifica con `%`):

```
SELECT idAutor, nombrePila, apellidoPaterno
   FROM autores
   WHERE apellidoPaterno LIKE '_o%'
```

La consulta anterior produce la fila que se muestra en la figura 25.14, ya que sólo un autor en nuestra base de datos tiene un apellido paterno que contiene la letra `o` como su segunda letra.

idAutor	nombrePila	apellidoPaterno
3	Andrew	Goldberg

Figura 25.14 | El único autor de la tabla `autores` cuyo apellido paterno contiene `o` como la segunda letra.

25.4.3 La cláusula ORDER BY

Las filas en el resultado de una consulta pueden ordenarse en forma ascendente o descendente, mediante el uso de la **cláusula** ORDER BY opcional. La forma básica de una consulta con una cláusula ORDER BY es:

SELECT *nombreDeColumna1, nombreDeColumna2 , …* FROM *nombreDeTabla* ORDER BY *columna* ASC
SELECT *nombreDeColumna1, nombreDeColumna2 , …* FROM *nombreDeTabla* ORDER BY *columna* DESC

en donde ASC especifica el orden ascendente (de menor a mayor), DESC especifica el orden descendente (de mayor a menor) y *columna* especifica la columna en la cual se basa el ordenamiento. Por ejemplo, para obtener la lista de autores en orden ascendente por apellido paterno (figura 25.15), utilice la siguiente consulta:

```
SELECT idAutor, nombrePila, apellidoPaterno
    FROM autores
    ORDER BY apellidoPaterno ASC
```

Observe que el orden predeterminado es ascendente, por lo que ASC es opcional. Para obtener la misma lista de autores en orden descendente por apellido paterno (figura 25.16), utilice la siguiente consulta:

```
SELECT idAutor, nombrePila, apellidoPaterno
    FROM autores
    ORDER BY apellidoPaterno DESC
```

Pueden usarse varias columnas para ordenar mediante una cláusula ORDER BY, de la siguiente forma:

ORDER BY *columna1 tipoDeOrden*, *columna2 tipoDeOrden*, …

en donde *tipoDeOrden* puede ser ASC o DESC. Observe que el *tipoDeOrden* no tiene que ser idéntico para cada columna. La consulta:

```
SELECT idAutor, nombrePila, apellidoPaterno
    FROM autores
    ORDER BY apellidoPaterno, nombrePila
```

idAutor	nombrePila	apellidoPaterno
4	David	Choffnes
1	Harvey	Deitel
2	Paul	Deitel
3	Andrew	Goldberg

Figura 25.15 | Datos de ejemplo de la tabla autores ordenados en forma ascendente por la columna apellidoPaterno.

idAutor	nombrePila	apellidoPaterno
3	Andrew	Goldberg
1	Harvey	Deitel
2	Paul	Deitel
4	David	Choffnes

Figura 25.16 | Datos de ejemplo de la tabla autores ordenados en forma descendente por la columna apellidoPaterno.

ordena en forma ascendente todas las filas por apellido paterno, y después por nombre de pila. Si varias filas tienen el mismo valor de apellido paterno, se devuelven ordenadas por nombre de pila (figura 25.17).

Las cláusulas `WHERE` y `ORDER BY` pueden combinarse en una consulta. Por ejemplo, la consulta:

```
SELECT isbn, titulo, numeroEdicion, copyright, precio
   FROM titulos
   WHERE titulo LIKE '%How to Program'
   ORDER BY titulo ASC
```

devuelve el `isbn`, `titulo`, `numeroEdicion`, `copyright` y `precio` de cada libro en la tabla `titulos` que tenga un `titulo` que termine con `"How to Program"` y los ordena en forma ascendente, por `titulo`. El resultado de la consulta se muestra en la figura 25.18.

idAutor	nombrePila	apellidoPaterno
4	David	Choffner
1	Harvey	Deitel
2	Paul	Deitel
3	Andrew	Goldberg

Figura 25.17 | Datos de ejemplo de autores de la tabla `autores` ordenados de manera ascendente, por las columnas `apellidoPaterno` y `nombrePila`.

isbn	titulo	numeroEdicion	copyright
0132404168	C How to Program	5	2007
0131857576	C++ How to Program	5	2005
0131450913	Internet & World Wide Web How to Program	3	2004
0132222205	Java How to Program	7	2007
0131869000	Visual Basic 2005 How to Program	3	2006
013152539	Visual C# How to Program	2	2006

Figura 25.18 | Ejemplos de libros de la tabla `titulos` cuyos títulos terminan con `How to Program`, y están ordenados en forma ascendente por medio de la columna `titulo`.

25.4.4 Cómo fusionar datos de varias tablas: `INNER JOIN`

Los diseñadores de bases de datos a menudo dividen los datos en tablas separadas para asegurarse de no guardar información redundante. Por ejemplo, la base de datos `libros` tiene las tablas `autores` y `titulos`. Utilizamos una tabla `isbnAutor` para almacenar los datos de la relación entre los autores y sus correspondientes títulos. Si no separáramos esta información en tablas individuales, tendríamos que incluir la información del autor con cada entrada en la tabla `titulos`. Esto implicaría almacenar información duplicada sobre los autores en la base de datos, para quienes hayan escrito varios libros. A menudo es necesario fusionar datos de varias tablas en un solo resultado. Este proceso, que se le conoce como unir las tablas, se especifica mediante un operador `INNER JOIN` en la consulta. Un operador `INNER JOIN` fusiona las filas de dos tablas al relacionar los valores en columnas que sean comunes para las dos tablas. La forma básica de un `INNER JOIN` es:

```
SELECT nombreDeColumna1, nombreDeColumna2, …
FROM tabla1
```

```
INNER JOIN tabla2
    ON tabla1.nombreDeColumna = tabla2.nombreDeColumna
```

La **cláusula ON** de `INNER JOIN` especifica las columnas de cada tabla que se comparan para determinar cuáles filas se fusionan. Por ejemplo, la siguiente consulta produce una lista de autores acompañada por los ISBNs para los libros escritos por cada autor.

```
SELECT nombrePila, apellidoPaterno, isbn
FROM autores
INNER JOIN isbnAutor
    ON autores.idAutor = isbnAutor.idAutor
ORDER BY apellidoPaterno, nombrePila
```

La consulta fusiona los datos de las columnas `nombrePila` y `apellidoPaterno` de la tabla `autores` con la columna `isbn` de la tabla `isbnAutor`, ordenando el resultado en forma ascendente por `apellidoPaterno` y `nombrePila`. Observe el uso de la sintaxis *nombreDeTabla.nombreDeColumna* en la cláusula `ON`. Esta sintaxis (a la que se le conoce como **nombre calificado**) especifica las columnas de cada tabla que deben compararse para unir las tablas. La sintaxis "*nombreDeTabla*." es requerida si las columnas tienen el mismo nombre en ambas tablas. La misma sintaxis puede utilizarse en cualquier consulta para diferenciar entre columnas de tablas distintas que tengan el mismo nombre. En algunos sistemas pueden utilizarse nombres de tablas calificados con el nombre de la base de datos para realizar consultas entre varias bases de datos. Como siempre, la consulta puede contener una cláusula `ORDER BY`. En la figura 25.19 se muestra una parte de los resultados de la consulta anterior, ordenados por `apellidoPaterno` y `nombrePila`. [*Nota:* para ahorrar espacio dividimos el resultado de la consulta en dos columnas, cada una de las cuales contiene a las columnas `nombrePila`, `apellidoPaterno` e `isbn`].



Observación de ingeniería de software 25.3

Si una instrucción de SQL incluye columnas de varias tablas que tengan el mismo nombre, en la instrucción se debe anteponer a los nombres de esas columnas los nombres de sus tablas y el operador punto (por ejemplo, `autores.idAutor`*).*



Error común de programación 25.5

Si no se califican los nombres para las columnas que tengan el mismo nombre en dos o más tablas se produce un error.

nombrePila	apellidoPaterno	isbn	nombrePila	apellidoPaterno	isbn
David	Choffnes	0131828274	Paul	Deitel	0131525239
Harvey	Deitel	0131525239	Paul	Deitel	0132404168
Harvey	Deitel	0132404168	Paul	Deitel	0131869000
Harvey	Deitel	0131869000	Paul	Deitel	0132222205
Harvey	Deitel	0132222205	Paul	Deitel	0131450913
Harvey	Deitel	0131450913	Paul	Deitel	0131525239
Harvey	Deitel	0131525239	Paul	Deitel	0131857576
Harvey	Deitel	0131857576	Paul	Deitel	0131828274
Harvey	Deitel	0131828274	Andrew	Goldberg	0131450913

Figura 25.19 | Ejemplo de datos de autores e ISBNs para los libros que han escrito, en orden ascendente por `apellidoPaterno` y `nombrePila`.

25.4.5 La instrucción INSERT

La instrucción INSERT inserta una fila en una tabla. La forma básica de esta instrucción es:

```
INSERT INTO nombreDeTabla ( nombreDeColumna1, nombreDeColumna2, …, nombreDeColumnaN )
   VALUES ( valor1, valor2, …, valorN )
```

en donde *nombreDeTabla* es la tabla en la que se va a insertar la fila. El *nombreDeTabla* va seguido de una lista separada por comas de nombres de columnas entre paréntesis (esta lista no es requerida si la operación INSERT especifica un valor para cada columna de la tabla en el orden correcto). La lista de nombres de columnas va seguida por la palabra clave VALUES de SQL, y una lista separada por comas de valores entre paréntesis. Los valores especificados aquí deben coincidir con las columnas especificadas después del nombre de la tabla, tanto en orden como en tipo (por ejemplo, si *nombreDeColumna1* se supone que debe ser la columna nombrePila, entonces *valor1* debe ser una cadena entre comillas sencillas que represente el nombre de pila). Siempre debemos enlistar explícitamente las columnas al insertar filas. Si el orden de las columnas cambia en la tabla, al utilizar solamente VALUES se puede provocar un error. La instrucción INSERT:

```
INSERT INTO autores ( nombrePila, apellidoPaterno )
   VALUES ( 'Alejandra', 'Villarreal' )
```

inserta una fila en la tabla autores. La instrucción indica que se proporcionan valores para las columnas nombrePila y apellidoPaterno. Los valores correspondientes son 'Alejandra' y 'Villarreal'. No especificamos un idAutor en este ejemplo, ya que idAutor es una columna autoincrementada en la tabla autores. Para cada fila que se agrega a esta tabla, MySQL asigna un valor de idAutor único que es el siguiente valor en la secuencia autoincrementada (por ejemplo: 1, 2, 3 y así sucesivamente). En este caso, Alejandra Villarreal recibiría el número 5 para idAutor. En la figura 25.20 se muestra la tabla autores después de la operación INSERT. [*Nota:* no todos los sistemas de administración de bases de datos soportan las columnas autoincrementadas. Consulte la documentación de su DBMS para encontrar alternativas a las columnas autoincrementadas].

idAutor	nombrePila	apellidoPaterno
1	Harvey	Deitel
2	Paul	Deitel
3	Andrew	Goldberg
4	David	Choffnes
5	Alejandra	Villarreal

Figura 25.20 | Datos de ejemplo de la tabla autores después de una operación INSERT.



Error común de programación 25.6

Por lo general, es un error especificar un valor para una columna que se autoincrementa.



Error común de programación 25.7

Las instrucciones de SQL utilizan el carácter de comilla sencilla (') como delimitador para las cadenas. Para especificar una cadena que contenga una comilla sencilla (como O'Malley) en una instrucción de SQL, la cadena debe tener dos comillas sencillas en la posición en la que aparezca el carácter de comilla sencilla en la cadena (por ejemplo, 'O''Malley'*). El primero de los dos caracteres de comilla sencilla actúa como carácter de escape para el segundo. Si no se utiliza el carácter de escape en una cadena que sea parte de una instrucción de SQL, se produce un error de sintaxis de SQL.*

25.4.6 La instrucción UPDATE

Una instrucción UPDATE modifica los datos en una tabla. La forma básica de la instrucción UPDATE es:

```
UPDATE nombreDeTabla
    SET nombreDeColumna1 = valor1, nombreDeColumna2 = valor2, …, nombreDeColumnaN = valorN
    WHERE criterios
```

en donde *nombreDeTabla* es la tabla que se va a actualizar. El *nombreDeTabla* va seguido por la palabra clave `SET` y una lista separada por comas de los pares nombre/valor de las columnas, en el formato *nombreDeColumna*=*valor*. La cláusula `WHERE` opcional proporciona los criterios que determinan cuáles filas se van a actualizar. Aunque no es obligatoria, la cláusula `WHERE` se utiliza comúnmente, a menos que se vaya a realizar un cambio en todas las filas. La instrucción `UPDATE`:

```
UPDATE autores
    SET apellidoPaterno = 'Garcia'
    WHERE apellidoPaterno = 'Villarreal' AND nombrePila = 'Alejandra'
```

actualiza una fila en la tabla `autores`. La instrucción indica que `apellidoPaterno` recibirá el valor `Garcia` para la fila en la que `apellidoPaterno` sea igual a `Villarreal` y `nombrePila` sea igual a `Alejandra`. [*Nota:* si hay varias filas con el mismo nombre de pila "Alejandra" y el apellido paterno "Villarreal", esta instrucción modificará a todas esas filas para que tengan el apellido paterno "Garcia"]. Si conocemos el `idAutor` desde antes de realizar la operación `UPDATE` (tal vez porque lo hayamos buscado con anterioridad), la cláusula `WHERE` se puede simplificar de la siguiente manera:

```
WHERE idAutor = 5
```

En la figura 25.21 se muestra la tabla `autores` después de realizar la operación `UPDATE`.

idAutor	nombrePila	apellidoPaterno
1	Harvey	Deitel
2	Paul	Deitel
3	Andrew	Goldberg
4	David	Choffnes
5	Alejandra	Garcia

Figura 25.21 | Datos de ejemplo de la tabla `autores` después de una operación `UPDATE`.

25.4.7 La instrucción DELETE

Una instrucción `DELETE` de SQL elimina filas de una tabla. La forma básica de una instrucción `DELETE` es:

```
DELETE FROM nombreDeTabla WHERE criterios
```

en donde *nombreDeTabla* es la tabla de la que se van a eliminar filas. La cláusula `WHERE` opcional especifica los criterios utilizados para determinar cuáles filas eliminar. Si se omite esta cláusula, se eliminan todas las filas de la tabla. La instrucción `DELETE`:

```
DELETE FROM autores
    WHERE apellidoPaterno = 'Garcia' AND nombrePila = 'Alejandra'
```

elimina la fila de Alejandra Garcia en la tabla `autores`. Si conocemos el `idAutor` desde antes de realizar la operación `DELETE`, la cláusula `WHERE` puede simplificarse de la siguiente manera:

```
WHERE idAutor = 5
```

En la figura 25.22 se muestra la tabla `autores` después de realizar la operación `DELETE`.

idAutor	nombrePila	apellidoPaterno
1	Harvey	Deitel
2	Paul	Deitel
3	Andrew	Goldberg
4	David	Choffnes

Figura 25.22 | Datos de ejemplo de la tabla `autores` después de una operación `DELETE`.

25.5 Instrucciones para instalar MySQL y MySQL Connector/J

MySQL 5.0 Community Edition es un sistema de administración de bases de datos de código fuente abierto que se ejecuta en muchas plataformas, incluyendo Windows, Solaris, Linux y Macintosh. En el sitio `www.mysql.com` encontrará toda la información acerca de MySQL. Los ejemplos en las secciones 25.8 y 25.9 manipulan bases de datos de MySQL.

Instalación de MySQL

Para instalar MySQL Community Edition:

1. Para aprender acerca de los requerimientos de instalación para su plataforma, visite el sitio `dev.mysql.com/doc/refman/5.0/en/general-installation-issues.html` (para ver esta información en español, visite el sitio `dev.mysql.com/doc/refman/5.0/es/general-installation-issues.html`).
2. Visite `dev.mysql.com/downloads/mysql/5.0.html` y descargue el instalador para su plataforma. Para los ejemplos de MySQL en este capítulo, sólo necesita el paquete Windows Essentials en Microsoft Windows, o el paquete Standard en la mayoría de las otras plataformas. [*Nota:* para estas instrucciones, vamos a suponer que está utilizando Microsoft Windows. En el sitio `dev.mysql.com/doc/refman/5.0/en/installing.html` (o `dev.mysql.com/doc/refman/5.0/es/installing.html` en español) encontrará las instrucciones completas de instalación para las otras plataformas].
3. Haga doble clic en el archivo `mysql-essential-5.0.27-win32.msi` para iniciar el instalador. [*Nota:* el nombre de este archivo puede diferir, dependiendo de la versión actual de MySQL 5.0].
4. Seleccione la opción **Typical** (**Típica**) en **Setup Type** (**Tipo de instalación**) y haga clic en **Next >** (**Siguiente**). Después haga clic en **Install** (**Instalar**).

Cuando termine la instalación, el programa le pedirá que configure una cuenta en MySQL.com. Si no desea hacerlo, seleccione **Skip Sign-up** (**Omitir registro**) y haga clic en **Next >** (**Siguiente**). Después de completar el proceso de registro o de omitirlo, puede configurar el servidor de MySQL. Haga clic en **Finish** (**Terminar**) para iniciar el **MySQL Server Instance Configuration Wizard** (**Asistente para la configuración de una instancia de MySQL Server**). Para configurar el servidor:

1. Haga clic en **Next >** (**Siguiente**); después seleccione **Standard Configuration** (**Configuración estándar**) y haga clic en **Next >** otra vez.
2. Tiene la opción de instalar MySQL como servicio Windows, lo cual permite al servidor de MySQL empezar a ejecutarse automáticamente cada vez que su sistema se inicie. Para nuestros ejemplos esto no es necesario, por lo que puede desactivar la opción **Install as a Windows Service** (**Instalar como servicio Windows**), y después haga clic en la opción **Include Bin Directory in Windows PATH** (**Incluir directorio Bin en la ruta PATH de Windows**). Esto le permitirá usar los comandos de MySQL en el Símbolo del sistema de Windows.
3. Haga clic en **Next >** y después en **Execute** (**Ejecutar**) para llevar a cabo la configuración del servidor.
4. Haga clic en **Finish** (**Terminar**) para cerrar el asistente.

Instalación de MySQL Connector/J

Para usar MySQL con JDBC, también necesita instalar **MySQL Connector/J** (la J representa a Java): un controlador de JDBC que permite a los programas usar JDBC para interactuar con MySQL. Puede descargar MySQL Connector/J de

```
dev.mysql.com/downloads/connector/j/5.0.html
```

La documentación para Connector/J se encuentra en `dev.mysql.com/doc/connector/j/en/connector-j.html`. Al momento de escribir este libro, la versión actual disponible en forma general de MySQL Connector/J es 5.0.4. Para instalar MySQL Connector/J:

1. Descargue el archivo `mysql-connector-java-5.0.4.zip`.
2. Abra `mysql-connector-java-5.0.4.zip` con un extractor de archivos, como WinZip (`www.winzip.com`). Extraiga su contenido en la unidad C:\. Esto creará un directorio llamado `mysql-connector-java-5.0.4`. La documentación para MySQL Connector/J está en el archivo `connector-j.pdf` en el subdirectorio `docs` de `mysql-connector-java-5.0.4`, o puede verla en línea, en el sitio `dev.mysql.com/doc/connector/j/en/connector-j.html`.

25.6 Instrucciones para establecer una cuenta de usuario de MySQL

Para que los ejemplos de MySQL se ejecuten correctamente, necesita configurar una cuenta de usuario que permita a los usuarios crear, eliminar y modificar una base de datos. Una vez instalado MySQL, siga los pasos que se muestran a continuación para configurar una cuenta de usuario (en estos pasos asumimos que MySQL está instalado en su directorio predeterminado):

1. Abra una ventana de Símbolo del sistema e inicie el servidor de bases de datos, ejecutando el comando `mysqld-nt.exe`. Observe que este comando no tiene salida; simplemente inicia el servidor MySQL. No cierre esta ventana, de lo contrario el servidor dejará de ejecutarse.
2. A continuación, inicie el monitor de MySQL para que pueda configurar una cuenta de usuario; abra otra ventana de Símbolo del sistema y ejecute el comando

   ```
   mysql –h localhost –u root
   ```

   La opción `–h` indica el host (computadora) en el que se está ejecutando el servidor MySQL; en este caso, es su equipo local (`localhost`). La opción `–u` indica la cuenta de usuario que se utilizará para iniciar sesión en el servidor; `root` es la cuenta de usuario predeterminada que se crea durante la instalación, para que usted pueda configurar el servidor. Una vez que inicie sesión, aparecerá un indicador `mysql>` en el que podrá escribir comandos para interactuar con el servidor MySQL.
3. En el indicador `mysql>`, escriba

   ```
   USE mysql;
   ```

   para seleccionar la base de datos incrustada, llamada `mysql`, la cual almacena información relacionada con el servidor, como las cuentas de usuario y sus privilegios para interactuar con el servidor. Observe que cada comando debe terminar con punto y coma. Para confirmar el comando, MySQL genera el mensaje "`Database changed.`" (La base de datos cambió).
4. A continuación, agregue la cuenta de usuario `jhtp7` a la base de datos incrustada `mysql`. Esta base de datos contiene una tabla llamada `user`, con columnas que representan el nombre del usuario, su contraseña y varios privilegios. Para crear la cuenta de usuario `jhtp7` con la contraseña `jhtp7`, ejecute los siguientes comandos desde el indicador `mysql>`:

   ```
   create user 'jhtp7'@'localhost' identified by 'jhtp7';

   grant select, insert, update, delete, create, drop, references,
      execute on *.* to 'jhtp7'@'localhost';
   ```

Esto crea el usuario `jhtp7` con los privilegios necesarios para crear las bases de datos utilizadas en este capítulo, y para manipular esas bases de datos. Por último,

5. Escriba el comando

```
exit;
```

para terminar el monitor MySQL.

25.7 Creación de la base de datos `libros` en MySQL

Para cada una de las bases de datos MySQL que veremos en este libro, proporcionamos una secuencia de comandos SQL en un archivo con la extensión `.sql` que configura la base de datos y sus tablas. Puede ejecutar estas secuencias de comandos en el monitor de MySQL. En el directorio de ejemplos de este capítulo, encontrará la secuencia de comandos SQL `libros.sql` para crear la base de datos `libros`. Para los siguientes pasos, vamos a suponer que el servidor MySQL (`mysqld-nt.exe`) sigue ejecutándose. Para ejecutar la secuencia de comandos `libros.sql`:

1. Abra una ventana de Símbolo del sistema y utilice el comando `cd` para cambiar al directorio en el que se encuentra la secuencia de comandos `libros.sql`.
2. Inicie el monitor de MySQL, escribiendo

```
mysql -h localhost -u jhtp7 -p
```

La opción `-p` hará que el monitor le pida la contraseña para el usuario `jhtp7`. Cuando ocurra esto, escriba la contraseña `jhtp7`.

3. Ejecute la secuencia de comandos, escribiendo

```
source libros.sql;
```

Esto creará un nuevo directorio llamado `libros` en el directorio `data` del servidor (en Windows, se encuentra en `C:\Archivos de programa\MySQL\MySQL Server 5.0\data` de manera predeterminada). Este nuevo directorio contiene la base de datos `libros`.

4. Escriba el comando

```
exit;
```

para terminar el monitor de MySQL. Ahora está listo para continuar con el primer ejemplo de JDBC.

25.8 Manipulación de bases de datos con JDBC

En esta sección presentaremos dos ejemplos. El primero le enseñará cómo conectarse a una base de datos y hacer consultas en ella. El segundo ejemplo le demostrará cómo mostrar en pantalla el resultado de la consulta.

25.8.1 Cómo conectarse y realizar consultas en una base de datos

El ejemplo de la figura 25.23 realiza una consulta simple en la base de datos `libros` para obtener toda la tabla `autores` y mostrar los datos. El programa muestra cómo conectarse a la base de datos, hacer una consulta en la misma y procesar el resultado. En la siguiente discusión presentaremos los aspectos clave del programa relacionados con JDBC. [*Nota:* en las secciones 25.5 a 25.7 se muestra cómo iniciar el servidor MySQL, configurar una cuenta de usuario y crear la base de datos `libros`. Estos pasos *deben* realizarse antes de ejecutar el programa de la figura 25.23].

```java
// Fig. 25.23: MostrarAutores.java
// Muestra el contenido de la tabla autores.
import java.sql.Connection;
import java.sql.Statement;
```

Figura 25.23 | Cómo mostrar el contenido de la tabla `autores`. (Parte 1 de 3).

```
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.ResultSetMetaData;
import java.sql.SQLException;

public class MostrarAutores
{
   // nombre del controlador de JDBC y URL de la base de datos
   static final String CONTROLADOR = "com.mysql.jdbc.Driver";
   static final String URL_BASEDATOS = "jdbc:mysql://localhost/libros";

   // inicia la aplicación
   public static void main( String args[] )
   {
      Connection conexion = null; // maneja la conexión
      Statement instruccion = null; // instrucción de consulta
      ResultSet conjuntoResultados = null; // maneja los resultados

      // se conecta a la base de datos libros y realiza una consulta
      try
      {
         // carga la clase controlador
         Class.forName( CONTROLADOR );

         // establece la conexión a la base de datos
         conexion =
            DriverManager.getConnection( URL_BASEDATOS, "jhtp7", "jhtp7" );

         // crea objeto Statement para consultar la base de datos
         instruccion = conexion.createStatement();

         // consulta la base de datos
         conjuntoResultados = instruccion.executeQuery(
            "SELECT IDAutor, nombrePila, apellidoPaterno FROM autores" );

         // procesa los resultados de la consulta
         ResultSetMetaData metaDatos = conjuntoResultados.getMetaData();
         int numeroDeColumnas = metaDatos.getColumnCount();
         System.out.println( "Tabla Autores de la base de datos Libros:\n" );

         for ( int i = 1; i <= numeroDeColumnas; i++ )
            System.out.printf( "%-8s\t", metaDatos.getColumnName( i ) );
         System.out.println();

         while ( conjuntoResultados.next() )
         {
            for ( int i = 1; i <= numeroDeColumnas; i++ )
               System.out.printf( "%-8s\t", conjuntoResultados.getObject( i ) );
            System.out.println();
         } // fin de while
      }  // fin de try
      catch ( SQLException excepcionSql )
      {
         excepcionSql.printStackTrace();
      } // fin de catch
      catch ( ClassNotFoundException noEncontroClase )
      {
         noEncontroClase.printStackTrace();
      } // fin de catch
```

Figura 25.23 | Cómo mostrar el contenido de la tabla `autores`. (Parte 2 de 3).

```
64         finally // asegura que conjuntoResultados, instruccion y conexion estén cerrados
65         {
66            try
67            {
68               conjuntoResultados.close();
69               instruccion.close();
70               conexion.close();
71            } // fin de try
72            catch ( Exception excepcion )
73            {
74               excepcion.printStackTrace();
75            } // fin de catch
76         } // fin de finally
77      } // fin de main
78   } // fin de la clase MostrarAutores
```

```
Tabla Autores de la base de datos Libros:

IDAutor        nombrePila        apellidoPaterno
1              Harvey            Deitel
2              Paul              Deitel
3              Andrew            Goldberg
4              David             Choffnes
```

Figura 25.23 | Cómo mostrar el contenido de la tabla `autores`. (Parte 3 de 3).

En las líneas 3 a 8 se importan las interfaces de JDBC y las clases del paquete `java.sql` que se utilizan en este programa. En la línea 13 se declara una constante de cadena para el controlador de la base de datos, y en la línea 14 se declara una constante de cadena para el URL de la base de datos. Estas constantes identifican el nombre de la base de datos a la que nos conectaremos, así como información acerca del protocolo utilizado por el controlador JDBC (que veremos en breve). El método `main` (líneas 17 a 77) se conecta a la base de datos `libros`, realiza una consulta en la base de datos, muestra el resultado de esa consulta y cierra la conexión a la base de datos.

El programa debe cargar el controlador de bases de datos para poder conectarse a la base de datos. En la línea 27 se utiliza el método `static` **`forName`** de la clase `Class` para cargar la clase para el controlador de bases de datos. Esta línea lanza una excepción verificada del tipo **`java.lang.ClassNotFoundException`** si el cargador de clases no puede localizar la clase del controlador. Para evitar esta excepción, necesitamos incluir el archivo `mysql-connector-java-5.0.4-bin.jar` (en el directorio `C:\mysql-connector-java-5.0.4`) en la ruta de clases del programa a la hora de ejecutarlo, como se muestra a continuación:

```
java –classpath
  .;c:\mysql-connector-java-5.0.4\mysql-connector-java-5.0.4-bin.jar
  MostrarAutores
```

En la ruta de clases del comando anterior, observe el punto (`.`) al principio de la información de la ruta de clases. Si se omite este punto, la JVM no buscará las clases en el directorio actual, y por ende, no encontrará el archivo de la clase `MostrarAutores`. También puede copiar el archivo `mysql-connector-java-5.0.4-bin.jar` al directorio `C:\Archivos de programa\Java\jdk1.6.0\jre\lib\ext`. Después de hacer esto, puede ejecutar la aplicación simplemente utilizando el comando `java MostrarAutores`.



Observación de ingeniería de software 25.4

La mayoría de los distribuidores de bases de datos proporcionan sus propios controladores de bases de datos JDBC, y muchos distribuidores independientes proporcionan también controladores JDBC. Para obtener más información acerca de los controladores de JDBC, visite el sitio Web sobre JDBC de Sun Microsystems en `servlet.java.sun.com/products/jdbc/drivers`.

En las líneas 30 y 31 de la figura 25.23 se crea un objeto `Connection` (paquete `java.sql`), el cual es referenciado mediante `conexion`. Un objeto que implementa a la interfaz `Connection` administra la conexión entre el programa de Java y la base de datos. Los objetos `Connection` permiten a los programas crear instrucciones de SQL para manipular bases de datos. El programa inicializa a `conexion` con el resultado de una llamada al método `static getConnection` de la clase `DriverManager` (paquete `java.sql`), el cual trata de conectarse a la base de datos especificada mediante su URL. El método `getConnection` recibe tres argumentos: un objeto `String` que especifica el URL de la base de datos, un objeto `String` que especifica el nombre de usuario y un objeto `String` que especifica la contraseña. El nombre de usuario y la contraseña se establecen en la sección 25.6. Si utilizó un nombre de usuario y contraseña distintos, necesita reemplazar el nombre de usuario (segundo argumento) y la contraseña (tercer argumento) que se pasan al método `getConnection` en la línea 31. El URL localiza la base de datos (posiblemente en una red o en el sistema de archivos local de la computadora). El URL `jdbc:mysql://localhost/libros` especifica el protocolo para la comunicación (`jdbc`), el **subprotocolo** para la comunicación (`mysql`) y la ubicación de la base de datos (`//localhost/libros`, en donde `localhost` es el host que ejecuta el servidor MySQL y `libros` es el nombre de la base de datos). El subprotocolo `mysql` indica que el programa utiliza un subprotocolo específico de MySQL para conectarse a la base de datos MySQL. Si el objeto `DriverManager` no se puede conectar a la base de datos, el método `getConnection` lanza una excepción `SQLException` (paquete `java.sql`). En la figura 25.24 se enlistan los nombres de los controladores de JDBC y los formatos de URL de base de datos de varios RDBMSs populares.



Observación de ingeniería de software 25.5

La mayoría de los sistemas de administración de bases de datos requieren que el usuario inicie sesión para poder administrar el contenido de la base de datos. El método `getConnection` de `DriverManager` está sobrecargado con versiones que permiten al programa proporcionar el nombre de usuario y la contraseña para obtener acceso.

En la línea 34 se invoca el método `createStatement` de `Connection` para obtener un objeto que implemente a la interfaz `Statement` (paquete `java.sql`). El programa utiliza al objeto `Statement` para enviar instrucciones de SQL a la base de datos.

En las líneas 37 y 38 se utiliza el método `executeQuery` del objeto `Statement` para enviar una consulta que seleccione toda la información sobre los autores en la tabla `autores`. Este método devuelve un objeto que implementa a la interfaz `ResultSet`, y contiene el resultado de esta consulta. Los métodos de `ResultSet` permiten al programa manipular el resultado de la consulta.

En las líneas 41 a 54 se procesa el objeto `ResultSet`. En la línea 41 se obtienen los metadatos para el objeto `ResultSet` en forma de un objeto `ResultSetMetaData` (paquete `java.sql`). Los **metadatos** describen el contenido del objeto `ResultSet`. Los programas pueden usar metadatos mediante la programación, para obtener información acerca de los nombres y tipos de las columnas del objeto `ResultSet`. En la línea 42 se utiliza el método `ResultSetMetaData` de `getColumnCount` para obtener el número de columnas para el objeto `ResultSet`. En las líneas 45 y 46 se anexan los nombres de las columnas.

RDBMS	Formato de URL de base de datos
MySQL	`jdbc:mysql:`*//nombrehost:numeroPuerto/nombreBaseDatos*
ORACLE	`jdbc:oracle:thin:`*@nombrehost:numeroPuerto:nombreBaseDatos*
DB2	`jdbc:db2:`*nombrehost:numeroPuerto/nombreBaseDatos*
Java DB/Apache Derby	`jdbc:derby:`*nombreBaseDatos* (incrustado) `jdbc:derby:`*//nombrehost:numeroPuerto/nombreBaseDatos* (red)
Microsoft SQL Server	`jdbc:sqlserver:`*//nombrehost.numeroPuerto*;nombreBaseDatos=*nombreBaseDatos*
Sybase	`jdbc:sybase:`Tds:*nombrehost:numeroPuerto/nombreBaseDatos*

Figura 25.24 | Formatos populares de URL de base de datos.



Observación de ingeniería de software 25.6

Los metadatos permiten a los programas procesar el contenido de un objeto `ResultSet` en forma dinámica, cuando no se conoce de antemano la información detallada acerca del objeto `ResultSet`.

En las líneas 49 a 54 se muestran los datos en cada fila del objeto `ResultSet`. Primero, el programa posiciona el cursor de `ResultSet` (que apunta a la fila que se está procesando) en la primera fila en el objeto `ResultSet`, mediante el método `next` (línea 49). Este método devuelve el valor `boolean true` si el cursor puede posicionarse en la siguiente fila; en caso contrario el método devuelve `false`.



Error común de programación 25.8

Inicialmente, un cursor `ResultSet` se posiciona antes de la primera fila. Si trata de acceder al contenido de un objeto `ResultSet` antes de posicionar el cursor `ResultSet` en la primera fila con el método `next`, se produce una excepción `SQLException`.

Si hay filas en el objeto `ResultSet`, en la línea 52 se extrae el contenido de una columna en la fila actual. Al procesar un objeto `ResultSet`, es posible extraer cada columna de este objeto como un tipo de Java específico. De hecho, el método `getColumnType` de `ResultSetMetaData` devuelve un valor entero constante de la clase `Types` (paquete `java.sql`), indicando el tipo de una columna especificada. Los programas pueden utilizar estos valores en una estructura `switch` para invocar métodos de `ResultSet` que devuelvan los valores de las columnas como tipos de Java apropiados. Si el tipo de una columna es `Types.INTEGER`, el método `getInt` de `ResultSet` devuelve el valor de la columna como un `int`. Los métodos *obtener (get)* de `ResultSet` generalmente reciben como argumento un número de columna (como un valor `int`) o un nombre de columna (como un valor `String`), indicando cuál valor de la columna se va a obtener. Visite la página

```
java.sun.com/javase/6/docs/technotes/guides/jdbc/getstart/
   GettingStartedTOC.fm.html
```

para obtener las asociaciones detalladas de los tipos de datos SQL y los tipos de Java, y para determinar el método de `ResultSet` apropiado a llamar a cada tipo de datos SQL.



Tip de rendimiento 25.1

Si una consulta especifica las columnas exactas a seleccionar de la base de datos, entonces el objeto `ResultSet` contendrá las columnas en el orden especificado. En este caso, es más eficiente utilizar el número de columna para obtener su valor que utilizar su nombre. El número de columna proporciona un acceso directo a la columna especificada. Si se usa el nombre, se requiere una búsqueda lineal de los nombres de columna para localizar la apropiada.

Por cuestiones de simpleza, en este ejemplo trataremos a cada valor como un objeto `Object`. El programa obtiene el valor de cada columna con el método `getObject` de `ResultSet` (línea 52) e imprime la representación `String` del objeto `Object`. Observe que, a diferencia de los índices de arreglos que empiezan en 0, los números de columna de `ResultSet` empiezan en 1. El bloque `finally` (líneas 64 a 76) cierra los objetos `ResultSet` (línea 68), `Statement` (línea 69) y el objeto `Connection` (línea 70) de la base de datos. [*Nota:* en las líneas 68 a 70 se lanzarán excepciones `NullPointerException` si los objetos `ResultSet`, `Statement` o `Connection` no se crearon en forma apropiada. En código de producción, debemos comprobar las variables que se refieren a estos objetos, para ver si son `null` antes de llamar a `close`].



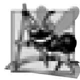
Error común de programación 25.9

Si se especifica el número de columna 0 cuando se van a obtener valores de un objeto `ResultSet`, se produce una excepción `SQLException`.



Error común de programación 25.10

Al tratar de manipular un objeto `ResultSet` después de cerrar el objeto `Statement` que lo creó, se produce una excepción `SQLException`. El programa descarta el objeto `ResultSet` cuando se cierra el objeto `Statement` correspondiente.



Observación de ingeniería de software 25.7

Cada objeto `Statement` puede abrir solamente un objeto `ResultSet` en un momento dado. Cuando un objeto `Statement` devuelve un nuevo objeto `ResultSet`, el objeto `Statement` cierra el objeto `ResultSet` anterior. Para utilizar varios objetos `ResultSet` en paralelo, se deben usar objetos `Statement` separados para devolver los objetos `ResultSet`.

25.8.2 Consultas en la base de datos `libros`

El siguiente ejemplo (figuras 25.25 y 25.28) permite al usuario introducir cualquier consulta en el programa. Este ejemplo muestra el resultado de una consulta en un objeto `JTable`, utilizando un objeto `TableModel` para proporcionar los datos del objeto `ResultSet` al objeto `JTable`. Un objeto `JTable` es un componente de la GUI de Swing que puede enlazarse a una base de datos para mostrar los resultados de una consulta. La clase `ResultSetTableModel` (figura 25.25) realiza la conexión a la base de datos por medio de un objeto `TableModel` y mantiene el objeto `ResultSet`. La clase `MostrarResultadosConsulta` (figura 25.28) crea la GUI y especifica una instancia de la clase `ResultSetTableModel` para proporcionar datos para el objeto `JTable`.

La clase `ResultSetTableModel`

La clase `ResultSetTableModel` (figura 25.25) extiende a la clase `AbstractTableModel` (paquete `javax.swing.table`), la cual implementa a la interfaz `TableModel`. La clase `ResultSetTableModel` sobrescribe a los métodos `getColumnClass`, `getColumnCount`, `getColumnName`, `getRowCount` y `getValueAt` de `TableModel`. Las implementaciones predeterminadas de los métodos `isCellEditable` y `setValueAt` de `TableModel` (proporcionados por `AbstractTableModel`) no se sobrescriben, ya que este ejemplo no soporta la capacidad de editar las celdas del objeto `JTable`. Tampoco se sobrescriben las implementaciones predeterminadas de los métodos `addTableModelListener` y `removeTableModelListener` de `TableModel` (proporcionados por `AbstractTableModel`), ya que las implementaciones de estos métodos de `AbstractTableModel` agregan y eliminan apropiadamente los componentes de escucha de eventos.

```java
// Fig. 25.25: ResultSetTableModel.java
// Un objeto TableModel que suministra datos ResultSet a un objeto JTable.
import java.sql.Connection;
import java.sql.Statement;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.ResultSetMetaData;
import java.sql.SQLException;
import javax.swing.table.AbstractTableModel;

// las filas y columnas del objeto ResultSet se cuentan desde 1 y
// las filas y columnas del objeto JTable se cuentan desde 0. Al procesar
// filas o columnas de ResultSet para usarlas en un objeto JTable, es
// necesario sumar 1 al número de fila o columna para manipular
// la columna apropiada del objeto ResultSet (es decir, la columna 0 de JTable
// es la columna 1 de ResultSet y la fila 0 de JTable es la fila 1 de ResultSet).
public class ResultSetTableModel extends AbstractTableModel
{
   private Connection conexion;
   private Statement instruccion;
   private ResultSet conjuntoResultados;
   private ResultSetMetaData metaDatos;
   private int numeroDeFilas;

   // lleva la cuenta del estado de la conexión a la base de datos
   private boolean conectadoABaseDatos = false;
```

Figura 25.25 | Un objeto `TableModel` que suministra datos `ResultSet` a un objeto `JTable`. (Parte 1 de 4).

```

   // el constructor inicializa conjuntoResultados y obtiene su objeto de metadatos;
   // determina el número de filas
   public ResultSetTableModel( String controlador,String url, String nombreusuario,
        String contrasenia, String consulta )
      throws SQLException, ClassNotFoundException
   {
      Class.forName( controlador );
      // se conecta a la base de datos
      conexion = DriverManager.getConnection( url, nombreusuario, contrasenia );

      // crea objeto Statement para consultar la base de datos
      instruccion = conexion.createStatement(
         ResultSet.TYPE_SCROLL_INSENSITIVE,
         ResultSet.CONCUR_READ_ONLY );

      // actualiza el estado de la conexión a la base de datos
      conectadoABaseDatos = true;

      // establece consulta y la ejecuta
      establecerConsulta( consulta );
   } // fin del constructor ResultSetTableModel

   // obtiene la clase que representa el tipo de la columna
   public Class getColumnClass( int columna ) throws IllegalStateException
   {
      // verifica que esté disponible la conexión a la base de datos
      if ( !conectadoABaseDatos )
         throw new IllegalStateException( "No hay conexion a la base de datos" );

      // determina la clase de Java de la columna
      try
      {
         String nombreClase = metaDatos.getColumnClassName( columna + 1 );

         // devuelve objeto Class que representa a nombreClase
         return Class.forName( nombreClase );
      } // fin de try
      catch ( Exception excepcion )
      {
         excepcion.printStackTrace();
      } // fin de catch

      return Object.class; // si ocurren problemas en el código anterior, asume el tipo
      Object
   } // fin del método getColumnClass

   // obtiene el número de columnas en el objeto ResultSet
   public int getColumnCount() throws IllegalStateException
   {
      // verifica que esté disponible la conexión a la base de datos
      if ( !conectadoABaseDatos )
         throw new IllegalStateException( "No hay conexión a la base de datos" );

      // determina el número de columnas
      try
      {
         return metaDatos.getColumnCount();
```

Figura 25.25 | Un objeto `TableModel` que suministra datos `ResultSet` a un objeto `JTable`. (Parte 2 de 4).

```
      } // fin de try
      catch ( SQLException excepcionSql )
      {
         excepcionSql.printStackTrace();
      } // fin de catch

      return 0; // si ocurren problemas en el código anterior, devuelve 0 para el
      número de columnas
   } // fin del método getColumnCount

   // obtiene el nombre de una columna específica en el objeto ResultSet
   public String getColumnName( int columna ) throws IllegalStateException
   {
      // verifica que esté disponible la conexión a la base de datos
      if ( !conectadoABaseDatos )
         throw new IllegalStateException( "No hay conexion a la base de datos" );

      // determina el nombre de la columna
      try
      {
         return metaDatos.getColumnName( columna + 1 );
      } // fin de try
      catch ( SQLException excepcionSql )
      {
         excepcionSql.printStackTrace();
      } // fin de catch

      return ""; // si hay problemas, devuelve la cadena vacía para el nombre de la
      columna
   } // fin del método getColumnName

   // devuelve el número de filas en el objeto ResultSet
   public int getRowCount() throws IllegalStateException
   {
      // verifica que esté disponible la conexión a la base de datos
      if ( !conectadoABaseDatos )
         throw new IllegalStateException( "No hay conexion a la base de datos" );

      return numeroDeFilas;
   } // fin del método getRowCount

   // obtiene el valor en la fila y columna específicas
   public Object getValueAt( int fila, int columna )
      throws IllegalStateException
   {
      // verifica que esté disponible la conexión a la base de datos
      if ( !conectadoABaseDatos )
         throw new IllegalStateException( "No hay conexion a la base de datos" );

      // obtiene un valor en una fila y columna especificadas del objeto ResultSet
      try
      {
         conjuntoResultados.absolute( fila + 1 );
         return conjuntoResultados.getObject( columna + 1 );
      } // fin de try
      catch ( SQLException excepcionSql )
      {
         excepcionSql.printStackTrace();
```

Figura 25.25 | Un objeto `TableModel` que suministra datos `ResultSet` a un objeto `JTable`. (Parte 3 de 4).

```
140          } // fin de catch
141 
142          return ""; // si hay problemas, devuelve el objeto cadena vacía
143       } // fin del método getValueAt
144 
145       // establece nueva cadena de consulta en la base de datos
146       public void establecerConsulta( String consulta )
147          throws SQLException, IllegalStateException
148       {
149          // verifica que esté disponible la conexión a la base de datos
150          if ( !conectadoABaseDatos )
151             throw new IllegalStateException( "No hay conexion a la base de datos" );
152 
153          // especifica la consulta y la ejecuta
154          conjuntoResultados = instruccion.executeQuery( consulta );
155 
156          // obtiene metadatos para el objeto ResultSet
157          metaDatos = conjuntoResultados.getMetaData();
158 
159          // determina el número de filas en el objeto ResultSet
160          conjuntoResultados.last(); // avanza a la última fila
161          numeroDeFilas = conjuntoResultados.getRow(); // obtiene el número de fila
162 
163          // notifica al objeto JTable que el modelo ha cambiado
164          fireTableStructureChanged();
165       } // fin del método establecerConsulta
166 
167       // cierra objetos Statement y Connection
168       public void desconectarDeBaseDatos()
169       {
170          if ( conectadoABaseDatos )
171          {
172             // cierra objetos Statement y Connection
173             try
174             {
175                conjuntoResultados.close();
176                instruccion.close();
177                conexion.close();
178             } // fin de try
179             catch ( SQLException excepcionSql )
180             {
181                excepcionSql.printStackTrace();
182             } // fin de catch
183             finally  // actualiza el estado de la conexión a la base de datos
184             {
185                conectadoABaseDatos = false;
186             } // fin de finally
187          } // fin de if
188       } // fin del método desconectarDeBaseDatos
189    }  // fin de la clase ResultSetTableModel
```

Figura 25.25 | Un objeto `TableModel` que suministra datos `ResultSet` a un objeto `JTable`. (Parte 4 de 4).

El constructor de `ResultSetTableModel` (líneas 30 a 48) acepta cinco argumentos `String`: el nombre del controlador de MySQL, el URL de la base de datos, el nombre de usuario, la contraseña y la consulta predeterminada a realizar. El constructor lanza cualquier excepción que ocurra en su cuerpo, de vuelta a la aplicación que creó el objeto `ResultSetTableModel`, para que la aplicación pueda determinar cómo manejar esa excepción (por ejemplo, reportar un error y terminar la aplicación). En la línea 34 se carga el controlador. En la línea 36 se esta-

blece una conexión a la base de datos. En las líneas 39 a 41 se invoca el método `createStatement` de `Connection` para obtener un objeto `Statement`. En este ejemplo utilizamos una versión del método `createStatement` que recibe dos argumentos: el tipo de conjunto de resultados y la concurrencia del conjunto de resultados. El **tipo de conjunto de resultados** (figura 25.26) especifica si el cursor del objeto `ResultSet` puede desplazarse en ambas direcciones o solamente hacia delante, y si el objeto `ResultSet` es susceptible a los cambios. Los objetos `ResultSet` que son susceptibles a los cambios los reflejan inmediatamente después de que éstos se realizan con los métodos de la interfaz `ResultSet`. Si un objeto `ResultSet` no es susceptible a los cambios, la consulta que produjo ese objeto `ResultSet` debe ejecutarse de nuevo para reflejar cualquier cambio realizado. La **concurrencia del conjunto de resultados** (figura 25.27) especifica si el objeto `ResultSet` puede actualizarse con los métodos de actualización de `ResultSet`. En este ejemplo utilizamos un objeto `ResultSet` que puede desplazarse, que no es susceptible a los cambios y es de sólo lectura. En la línea 47 se invoca nuestro método `establecerConsulta` de `ResultSetTableModel` (líneas 146 a 165) para ejecutar la consulta predeterminada.

Constante de tipo `static` de `ResultSet`	Descripción
`TYPE_FORWARD_ONLY`	Especifica que el cursor de un objeto `ResultSet` puede desplazarse solamente en dirección hacia delante (es decir, desde la primera hasta la última fila en el objeto `ResultSet`).
`TYPE_SCROLL_INSENSITIVE`	Especifica que el cursor de un objeto `ResultSet` puede desplazarse en cualquier dirección y que los cambios realizados al objeto `ResultSet` durante su procesamiento no se reflejarán en este objeto, a menos que el programa consulte la base de datos otra vez.
`TYPE_SCROLL_SENSITIVE`	Especifica que el cursor de un objeto `ResultSet` puede desplazarse en cualquier dirección y que los cambios realizados al objeto `ResultSet` durante su procesamiento se reflejarán inmediatamente en este objeto.

Figura 25.26 | Constantes de `ResultSet` para especificar el tipo del objeto `ResultSet`.

Constante de concurrencia `static` de `ResultSet`	Descripción
`CONCUR_READ_ONLY`	Especifica que un objeto `ResultSet` no puede actualizarse (es decir, los cambios en el contenido del objeto `ResultSet` no pueden reflejarse en la base de datos con los métodos *update* de `ResultSet`).
`CONCUR_UPDATABLE`	Especifica que un objeto `ResultSet` puede actualizarse (es decir, los cambios en el contenido del objeto `ResultSet` pueden reflejarse en la base de datos con los métodos *update* de `ResultSet`).

Figura 25.27 | Constantes de `ResultSet` para especificar las propiedades de los resultados.



Tip de portabilidad 25.3

Algunos controladores JDBC no soportan objetos `ResultSet` *desplazables. En dichos casos, generalmente el controlador devuelve un objeto* `ResultSet` *en el que el cursor puede moverse sólo hacia adelante. Para obtener más información, consulte la documentación de su controlador de bases de datos.*



Tip de portabilidad 25.4

Algunos controladores JDBC no soportan objetos `ResultSet` *actualizables. En dichos casos, generalmente el controlador devuelve un objeto* `ResultSet` *de sólo lectura. Para obtener más información, consulte la documentación de su controlador de bases de datos.*



Error común de programación 25.11

Al intentar actualizar un objeto `ResultSet` *cuando el controlador de base de datos no soporta objetos* `ResultSet` *actualizables se producen excepciones* `SQLException`*.*



Error común de programación 25.12

Al intentar mover el cursor hacia atrás mediante un objeto `ResultSet`*, cuando el controlador de base de datos no soporta el desplazamiento hacia atrás, se produce una excepción* `SQLException`*.*

El método `getColumnClass` (líneas 51 a 71) devuelve un objeto `Class` que representa a la superclase de todos los objetos en una columna específica. El objeto `JTable` utiliza esta información para configurar el desplegador de celdas y el editor de celdas predeterminados para esa columna en el objeto `JTable`. En la línea 60 se utiliza el método `getColumnClassName` de `ResultSetMetaData` para obtener el nombre de clase completamente calificado para la columna especificada. En la línea 63 se carga la clase y se devuelve el objeto `Class` correspondiente. Si ocurre una excepción, el bloque `catch` en las líneas 65 a 68 imprime un rastreo de la pila y en la línea 70 se devuelve `Object.class` (la instancia de `Class` que representa a la clase `Object`) como el tipo predeterminado. [*Nota:* en la línea 60 se utiliza el argumento `columna + 1`. Al igual que los arreglos, los números de fila y columna del objeto `JTable` se cuentan desde 0. Sin embargo, los números de fila y columna del objeto `ResultSet` se cuentan desde 1. Por lo tanto, al procesar filas o columnas de un objeto `ResultSet` para utilizarlas en un objeto `JTable`, es necesario sumar 1 al número de fila o de columna para manipular la fila o columna apropiada del objeto `ResultSet`].

El método `getColumnCount` (líneas 74 a 91) devuelve el número de columnas en el objeto `ResultSet` subyacente del modelo. En la línea 83 se utiliza el método `getColumnCount` de `ResultSetMetaData` para obtener el número de columnas en el objeto `ResultSet`. Si ocurre una excepción, el bloque `catch` en las líneas 85 a 88 imprime un rastreo de la pila y en la línea 93 se devuelve 0 como el número predeterminado de columnas.

El método `getColumnName` (líneas 94 a 111) devuelve el nombre de la columna en el objeto `ResultSet` subyacente del modelo. En la línea 103 se utiliza el método `getColumnName` de `ResultSetMetaData` para obtener el nombre de la columna del objeto `ResultSet`. Si ocurre una excepción, el bloque `catch` en las líneas 105 a 108 imprime un rastreo de la pila y en la línea 110 se devuelve la cadena vacía como el nombre predeterminado de la columna.

El método `getRowCount` (líneas 114 a 121) devuelve el número de filas en el objeto `ResultSet` subyacente del modelo. Cuando el método `establecerConsulta` (líneas 146 a 165) realiza una consulta, almacena el número de filas en la variable `numeroDeFilas`.

El método `getValueAt` (líneas 124 a 143) devuelve el objeto `Object` en una fila y columna específicas del objeto `ResultSet` subyacente del modelo. En la línea 134 se utiliza el método `absolute` de `ResultSet` para posicionar el cursor del objeto `ResultSet` en una fila específica. En la línea 135 se utiliza el método `getObject` de `ResultSet` para obtener el objeto `Object` en una columna específica de la fila actual. Si ocurre una excepción, el bloque `catch` en las líneas 137 a 140 imprime un rastreo de la pila y en la línea 142 se devuelve una cadena vacía como el valor predeterminado.

El método `establecerConsulta` (líneas 146 a 165) ejecuta la consulta que recibe como argumento para obtener un nuevo objeto `ResultSet` (línea 154). En la línea 157 se obtiene el objeto `ResultSetMetaData` para el nuevo objeto `ResultSet`. En la línea 160 se utiliza el método `last` de `ResultSet` para posicionar el cursor de `ResultSet` en la última fila del objeto `ResultSet`. [*Nota:* esto puede ser lento si la tabla contiene muchas filas]. En la línea 161 se utiliza el método `getRow` de `ResultSet` para obtener el número de fila de la fila actual en el objeto `ResultSet`. En la línea 164 se invoca el método `fireTableStructureChanged` (heredado de la clase `AbstractTableModel`) para notificar a cualquier objeto `JTable` que utilice a este objeto `ResultSetTableModel` como su modelo, que la estructura del modelo ha cambiado. Esto hace que el objeto `JTable` vuelva a llenar sus filas y columnas con los datos del nuevo objeto `ResultSet`. El método `establecerConsulta` lanza cualquier excepción que ocurra en su cuerpo, de vuelta a la aplicación que invocó a `establecerConsulta`.

El método `desconectarDeBaseDatos` (líneas 168 a 188) implementa un método de terminación apropiado para la clase `ResultSetTableModel`. Un diseñador de clases debe proporcionar un método `public` que los clientes de la clase deban invocar en forma explícita para liberar los recursos que haya utilizado un objeto. En este caso, el método `desconectarDeBaseDatos` cierra los objetos `ResultSet`, `Statement` y `Connection` (líneas 175 a 177),

los cuales se consideran recursos limitados. Los clientes de la clase `ResultSetTableModel` deben siempre invocar a este método cuando la instancia de esta clase ya no se necesite. Antes de liberar los recursos, en la línea 170 se verifica si la conexión ya está terminada. De no ser así, el método continúa. Observe que cada uno de los otros métodos en la clase lanzan una excepción `IllegalStateException` si el campo booleano `conectadoABaseDatos` es `false`. El método `desconectarDeBaseDatos` establece a `conectadoABaseDatos` en `false` (línea 185) para asegurarse que los clientes no utilicen una instancia de `ResultSetTableModel` después de que ésta haya sido eliminada. `IllegalStateExcepcion` es una excepción de las bibliotecas de Java que es apropiada para indicar esta condición de error.

La clase `MostrarResultadosConsulta`

La clase `MostrarResultadosConsulta` (figura 25.28) implementa la GUI de la aplicación e interactúa con el objeto `ResultSetTableModel` a través de un objeto `JTable`. Esta aplicación también demuestra las nuevas herramientas de ordenamiento y filtrado de `JTable`, que se introdujeron en Java SE 6.

En las líneas 27 a 30 y 33 se declaran el controlador de la base de datos, el URL, el nombre de usuario, la contraseña y la consulta predeterminada que se pasan al constructor de `ResultSetTableModel` para realizar la conexión inicial a la base de datos y ejecutar la consulta predeterminada. El constructor de `MostrarResultadosConsulta` (líneas 39 a 198) crea un objeto `ResultSetTableModel` y la GUI para la aplicación. En la línea 69 se crea el objeto `JTable` y se pasa un objeto `ResultSetTableModel` al constructor de `JTable`, el cual a su vez registra el objeto `JTable` como un componente de escucha para los eventos `TableModelEvent` que sean generados por el objeto `ResultSetTableModel`.

```
1   // Fig. 25.28: MostrarResultadosConsulta.java
2   // Muestra el contenido de la tabla Autores en la base de datos libros.
3   import java.awt.BorderLayout;
4   import java.awt.event.ActionListener;
5   import java.awt.event.ActionEvent;
6   import java.awt.event.WindowAdapter;
7   import java.awt.event.WindowEvent;
8   import java.sql.SQLException;
9   import java.util.regex.PatternSyntaxException;
10  import javax.swing.JFrame;
11  import javax.swing.JTextArea;
12  import javax.swing.JScrollPane;
13  import javax.swing.ScrollPaneConstants;
14  import javax.swing.JTable;
15  import javax.swing.JOptionPane;
16  import javax.swing.JButton;
17  import javax.swing.Box;
18  import javax.swing.JLabel;
19  import javax.swing.JTextField;
20  import javax.swing.RowFilter;
21  import javax.swing.table.TableRowSorter;
22  import javax.swing.table.TableModel;
23
24  public class MostrarResultadosConsulta extends JFrame
25  {
26     // URL de la base de datos, nombre de usuario y contraseña para JDBC
27     static final String CONTROLADOR = "com.mysql.jdbc.Driver";
28     static final String URL_BASEDATOS = "jdbc:mysql://localhost/libros";
29     static final String NOMBREUSUARIO = "jhtp7";
30     static final String CONTRASENIA = "jhtp7";
31
32     // la consulta predeterminada obtiene todos los datos de la tabla autores
33     static final String CONSULTA_PREDETERMINADA = "SELECT * FROM autores";
34
```

Figura 25.28 | Visualización del contenido de la base de datos `libros`. (Parte 1 de 5).

```
   private ResultSetTableModel modeloTabla;
   private JTextArea areaConsulta;

   // crea objeto ResultSetTableModel y GUI
   public MostrarResultadosConsulta()
   {
      super( "Visualizacion de los resultados de la consulta" );

      // crea objeto ResultSetTableModel y muestra la tabla de la base de datos
      try
      {
         // crea objeto TableModel para los resultados de la consulta SELECT * FROM
         autores
         modeloTabla = new ResultSetTableModel( CONTROLADOR, URL_BASEDATOS,
            NOMBREUSUARIO, CONTRASENIA, CONSULTA_PREDETERMINADA );

         // establece objeto JTextArea en el que el usuario escribe las consultas
         areaConsulta = new JTextArea( CONSULTA_PREDETERMINADA, 3, 100 );
         areaConsulta.setWrapStyleWord( true );
         areaConsulta.setLineWrap( true );

         JScrollPane scrollPane = new JScrollPane( areaConsulta,
            ScrollPaneConstants.VERTICAL_SCROLLBAR_AS_NEEDED,
            ScrollPaneConstants.HORIZONTAL_SCROLLBAR_NEVER );

         // establece objeto JButton para enviar las consultas
         JButton botonEnviar = new JButton( "Enviar consulta" );

         // crea objeto Box para manejar la colocación de areaConsulta y
         // botonEnviar en la GUI
         Box boxNorte = Box.createHorizontalBox();
         boxNorte.add( scrollPane );
         boxNorte.add( botonEnviar );

         // crea delegado de JTable para modeloTabla
         JTable tablaResultados = new JTable( modeloTabla );

         JLabel etiquetaFiltro = new JLabel( "Filtro:" );
         final JTextField textoFiltro = new JTextField();
         JButton botonFiltro = new JButton( "Aplicar filtro" );
         Box boxSur = boxNorte.createHorizontalBox();

         boxSur.add( etiquetaFiltro );
         boxSur.add( textoFiltro );
         boxSur.add( botonFiltro );

         // coloca los componentes de la GUI en el panel de contenido
         add( boxNorte, BorderLayout.NORTH );
         add( new JScrollPane( tablaResultados ), BorderLayout.CENTER );
         add( boxSur, BorderLayout.SOUTH );

         // crea componente de escucha de eventos para botonEnviar
         botonEnviar.addActionListener(

            new ActionListener()
            {
               // pasa la consulta al modelo de la tabla
               public void actionPerformed( ActionEvent evento )
               {
```

Figura 25.28 | Visualización del contenido de la base de datos `libros`. (Parte 2 de 5).

```
                     // realiza una nueva consulta
                     try
                     {
                        modeloTabla.establecerConsulta( areaConsulta.getText() );
                     } // fin de try
                     catch ( SQLException excepcionSql )
                     {
                        JOptionPane.showMessageDialog( null,
                           excepcionSql.getMessage(), "Error en base de datos",
                           JOptionPane.ERROR_MESSAGE );

                        // trata de recuperarse de una consulta inválida del usuario
                        // ejecutando la consulta predeterminada
                        try
                        {
                           modeloTabla.establecerConsulta( CONSULTA_PREDETERMINADA );
                           areaConsulta.setText( CONSULTA_PREDETERMINADA );
                        } // fin de try
                        catch ( SQLException excepcionSql2 )
                        {
                           JOptionPane.showMessageDialog( null,
                              excepcionSql2.getMessage(), "Error en base de datos",
                              JOptionPane.ERROR_MESSAGE );

                           // verifica que esté cerrada la conexión a la base de datos
                           modeloTabla.desconectarDeBaseDatos();

                           System.exit( 1 ); // termina la aplicación
                        } // fin de catch interior
                     } // fin de catch exterior
                  } // fin de actionPerformed
               }  // fin de la clase interna ActionListener
            ); // fin de la llamada a addActionListener

            final TableRowSorter< TableModel > sorter =
               new TableRowSorter< TableModel >( modeloTabla );
            tablaResultados.setRowSorter( sorter );
            setSize( 500, 250 ); // establece el tamaño de la ventana
            setVisible( true ); // muestra la ventana

            // crea componente de escucha para botonFiltro
            botonFiltro.addActionListener(
               new ActionListener()
               {
                  // pasa el texto del filtro al componente de escucha
                  public void actionPerformed( ActionEvent e )
                  {
                     String texto = textoFiltro.getText();

                     if ( texto.length() == 0 )
                        sorter.setRowFilter( null );
                     else
                     {
                        try
                        {
                           sorter.setRowFilter(
                              RowFilter.regexFilter( texto ) );
                        } // fin de try
                        catch ( PatternSyntaxException pse )
```

Figura 25.28 | Visualización del contenido de la base de datos `libros`. (Parte 3 de 5).

```
                   {
                      JOptionPane.showMessageDialog( null,
                         "Patron de exp reg incorrecto", "Patron de exp reg incorrecto",
                         JOptionPane.ERROR_MESSAGE );
                   } // fin de catch
                } // fin de else
             } // fin del método actionPerfomed
          } // fin de la clase interna anónima
       ); // fin de la llamada a addActionLister
    } // fin de try
    catch ( ClassNotFoundException noEncontroClase )
    {
       JOptionPane.showMessageDialog( null,
          "No se encontro controlador de base de datos", "No se encontro el
          controlador",
          JOptionPane.ERROR_MESSAGE );

       System.exit( 1 ); // termina la aplicación
    } // fin de catch
    catch ( SQLException excepcionSql )
    {
       JOptionPane.showMessageDialog( null, excepcionSql.getMessage(),
          "Error en base de datos", JOptionPane.ERROR_MESSAGE );

       // verifica que esté cerrada la conexión a la base de datos
       modeloTabla.desconectarDeBaseDatos();

       System.exit( 1 ); // termina la aplicación
    } // fin de catch

    // cierra la ventana cuando el usuario sale de la aplicación (se sobrescribe
    // el valor predeterminado de HIDE_ON_CLOSE)
    setDefaultCloseOperation( DISPOSE_ON_CLOSE );

    // verifica que esté cerrada la conexión a la base de datos cuando el usuario sale
    de la aplicación
    addWindowListener(

       new WindowAdapter()
       {
          // se desconecta de la base de datos y sale cuando se ha cerrado la ventana
          public void windowClosed( WindowEvent evento )
          {
             modeloTabla.desconectarDeBaseDatos();
             System.exit( 0 );
          } // fin del método windowClosed
       } // fin de la clase interna WindowAdapter
    ); // fin de la llamada a addWindowListener
 } // fin del constructor de MostrarResultadosConsulta

 // ejecuta la aplicación
 public static void main( String args[] )
 {
    new MostrarResultadosConsulta();
 } // fin de main
} // fin de la clase MostrarResultadosConsulta
```

Figura 25.28 | Visualización del contenido de la base de datos `libros`. (Parte 4 de 5).



Figura 25.28 | Visualización del contenido de la base de datos `libros`. (Parte 5 de 5).

En las líneas 86 a 125 se registra un manejador de eventos para el `botonEnviar` en el que el usuario hace clic para enviar una consulta a la base de datos. Cuando el usuario hace clic en el botón, el método `actionPerformed` (líneas 91 a 123) invoca al método `establecerConsulta` de la clase `ResultSetTableModel` para ejecutar la nueva consulta. Si la consulta del usuario falla (por ejemplo, debido a un error de sintaxis en la entrada del usuario), en las líneas 108 y 109 se ejecuta la consulta predeterminada. Si la consulta predeterminada también falla, podría haber un error más grave, por lo que en la línea 118 se asegura que se cierre la conexión a la base de datos y en la línea 120 se sale del programa. Las capturas de pantalla de la figura 25.28 muestran los resultados de dos consultas. La primera captura de pantalla muestra la consulta predeterminada, en la que se recuperan todos los datos de la tabla `autores` de la base de datos `libros`. La segunda captura de pantalla muestra una consulta que selecciona el nombre de pila y el apellido paterno de cada autor en la tabla `autores`, y combina esa información con el título y número de edición de la tabla `titulos`. Pruebe a introducir sus propias consultas en el área de texto y haga clic en el botón **Enviar consulta** para enviar la consulta.

A partir de Java SE 6, los objetos `JTable` ahora permiten a los usuarios ordenar filas en base a los datos en una columna específica. En las líneas 127 y 128 se utiliza la clase **`TableRowSorter`** (del paquete **`javax.swing.table`**) para crear un objeto que utilice nuestro objeto `ResultSetTableModel` para ordenar filas en el objeto `JTable` que muestre los resultados de la consulta. Cuando el usuario hace clic en el título de una columna

específica del objeto `JTable`, el objeto `TableRowSorter` interactúa con el objeto `TableModel` subyacente para reordenar las filas, con base en los datos en esa columna. En la línea 129 se utiliza el método **`setRowSorter`** del método `JTable` para especificar el objeto `TableRowSorter` para `tablaResultados`.

Otra de las nuevas características de los objetos `JTable` es la habilidad de ver subconjuntos de los datos del objeto `TableModel` subyacente. A esto se le conoce como filtrar los datos. En las líneas 134 a 160 se registra un manejador de eventos para el `botonFiltro` que el usuario oprime para filtrar los datos. En el método `action-Performed` (líneas 138 a 158), en la línea 140 se obtiene el texto de filtro. Si el usuario no especificó un texto de filtro, en la línea 143 se utiliza el método **`setRowFilter`** de `JTable` para eliminar cualquier filtro anterior, estableciendo el filtro en `null`. En caso contrario, en las líneas 148 y 149 se utiliza `setRowFilter` para especificar un objeto **`RowFilter`** (del paquete `javax.swing`) con base en la entrada del usuario. La clase `RowFilter` cuenta con varios métodos para crear filtros. El método `static` **`regexFilter`** recibe un objeto `String` que contiene un patrón de expresión regular como argumento, y un conjunto opcional de índices que especifican cuáles columnas se van a filtrar. Si no se especifican índices, entonces se busca en todas las columnas. En este ejemplo, el patrón de expresión regular es el texto que escribió el usuario. Una vez establecido el filtro, se actualizan los datos mostrados en el objeto `JTable` con base en el objeto `TableModel` filtrado.

25.9 La interfaz RowSet

En los ejemplos anteriores, aprendido a realizar consultas en una base de datos al establecer en forma explícita una conexión (`Connection`) a la base de datos, preparar una instrucción (`Statement`) para consultar la base de datos y ejecutar la consulta. En esta sección demostraremos la interfaz `RowSet`, la cual configura la conexión a la base de datos y prepara instrucciones de consulta en forma automática. La interfaz `RowSet` proporciona varios métodos *establecer (get)* que nos permiten especificar las propiedades necesarias para establecer una conexión (como el URL de la base de datos, el nombre de usuario y la contraseña) y crear un objeto `Statement` (como una consulta). `RowSet` también cuenta con varios métodos *obtener (get)* para devolver estas propiedades.

Hay dos tipos de objetos `RowSet`: conectados y desconectados. Un objeto **`RowSet` conectado** se conecta a la base de datos una sola vez, y permanece conectado hasta que termina la aplicación. Un objeto **`RowSet` desconectado** se conecta a la base de datos, ejecuta una consulta para obtener los datos de la base de datos y después cierra la conexión. Un programa puede cambiar los datos en un objeto `RowSet` desconectado, mientras éste se encuentre desconectado. Los datos modificados pueden actualizarse en la base de datos, después de que un objeto `RowSet` desconectado reestablece la conexión a la base de datos.

El paquete **`javax.sql.rowset`** contiene dos subinterfaces de `RowSet: JdbcRowSet` y `CachedRowSet`. **`JdbcRowSet`**, un objeto `RowSet` conectado, actúa como una envoltura alrededor de un objeto `ResultSet` y nos permite desplazarnos a través de las filas en el objeto `ResultSet`, y también actualizarlas. Recuerde que, de manera predeterminada, un objeto `ResultSet` no es desplazable y es de sólo lectura; debemos establecer explícitamente la constante de tipo del conjunto de resultados a `TYPE_SCROLL_INSENSITIVE` y establecer la constante de concurrencia del conjunto de resultados `CONCUR_UPDATABLE` para hacer que un objeto `ResultSet` sea desplazable y pueda actualizarse. Un objeto `JdbcRowSet` es desplazable y puede actualizarse de manera predeterminada. **`CachedRowSet`**, un objeto `RowSet` desconectado, coloca los datos de un objeto `ResultSet` en caché de memoria y los desconecta de la base de datos. Al igual que un objeto `JdbcRowSet`, un objeto `CachedRowSet` es desplazable y puede actualizarse de manera predeterminada. Un objeto `CachedRowSet` también es serializable, por lo que puede pasarse de una aplicación de Java a otra mediante una red, como Internet. Sin embargo, `CachedRowSet` tiene una limitación: la cantidad de datos que pueden almacenarse en memoria es limitada. El paquete `javax.sql.rowset` contiene otras tres subinterfaces de `RowSet`. Para obtener detalles de estas interfaces, visite `java.sun.com/javase/6/docs/technotes/guides/jdbc/getstart/rowsetImpl.html`.



Tip de portabilidad 25.5

Un objeto `RowSet` puede proporcionar capacidad de desplazamiento a los controladores que no tienen soporte para objetos `ResultSet` desplazables.

En la figura 25.29 se reimplementa el ejemplo de la figura 25.23, usando un objeto `RowSet`. En vez de establecer la conexión y crear un objeto `Statement` de manera explícita, en la figura 25.29 utilizamos un objeto `JdbcRowSet` para crear los objetos `Connection` y `Statement` de manera automática.

```
// Fig. 25.29: PruebaJdbcRowSet.java
// Visualización del contenido de la tabla autores, mediante el uso de JdbcRowSet.
import java.sql.ResultSetMetaData;
import java.sql.SQLException;
import javax.sql.rowset.JdbcRowSet;
import com.sun.rowset.JdbcRowSetImpl; // implementación de JdbcRowSet de Sun

public class PruebaJdbcRowSet
{
   // nombre del controlador de JDBC y URL de la base de datos
   static final String CONTROLADOR = "com.mysql.jdbc.Driver";
   static final String URL_BASEDATOS = "jdbc:mysql://localhost/libros";
   static final String NOMBREUSUARIO = "jhtp7";
   static final String CONTRASENIA = "jhtp7";

   // el constructor se conecta a la base de datos, la consulta, procesa
   // los resultados y los muestra en la ventana
   public PruebaJdbcRowSet()
   {
      // se conecta a la base de datos libros y la consulta
      try
      {
         Class.forName( CONTROLADOR );

         // especifica las propiedades del objeto JdbcRowSet
         JdbcRowSet rowSet = new JdbcRowSetImpl();
         rowSet.setUrl( URL_BASEDATOS ); // establece URL de la base de datos
         rowSet.setUsername( NOMBREUSUARIO ); // establece el nombre de usuario
         rowSet.setPassword( CONTRASENIA ); // establece contraseña
         rowSet.setCommand( "SELECT * FROM autores" ); // establece la consulta
         rowSet.execute(); // ejecuta la consulta

         // procesa los resultados de la consulta
         ResultSetMetaData metaDatos = rowSet.getMetaData();
         int numeroDeColumnas = metaDatos.getColumnCount();
         System.out.println( "Tabla Autores de la base de datos Libros:\n" );

         // muestra el encabezado del objeto RowSet
         for ( int i = 1; i <= numeroDeColumnas; i++ )
            System.out.printf( "%-8s\t", metaDatos.getColumnName( i ) );
         System.out.println();

         // muestra cada fila
         while ( rowSet.next() )
         {
            for ( int i = 1; i <= numeroDeColumnas; i++ )
               System.out.printf( "%-8s\t", rowSet.getObject( i ) );
            System.out.println();
         } // fin de while

         // cierra el objeto ResultSet subyacente, y los objetos Statement y Connection
         rowSet.close();
      } // fin de try
      catch ( SQLException excepcionSql )
      {
         excepcionSql.printStackTrace();
         System.exit( 1 );
      } // fin de catch
      catch ( ClassNotFoundException noEncontroClase )
```

Figura 25.29 | Visualización de la tabla `autores` mediante `JdbcRowSet`. (Parte 1 de 2).

```
60          {
61             noEncontroClase.printStackTrace();
62             System.exit( 1 );
63          } // fin de catch
64       } // fin del constructor de mostrarAutores
65
66       // inicia la aplicación
67       public static void main( String args[] )
68       {
69          PruebaJdbcRowSet aplicacion = new PruebaJdbcRowSet();
70       } // fin de main
71    } // fin de la clase PruebaJdbcRowSet
```

```
Tabla Autores de la base de datos Libros:

IDAutor       nombrePila       apellidoPaterno
1             Harvey           Deitel
2             Paul             Deitel
3             Andrew           Goldberg
4             David            Choffnes
```

Figura 25.29 | Visualización de la tabla `autores` mediante `JdbcRowSet`. (Parte 2 de 2).

El paquete **`com.sun.rowset`** proporciona las implementaciones de referencia de Sun de las interfaces en el paquete `javax.sql.rowset`. En la línea 26 se utiliza la implementación de referencia de Sun de la interfaz `JdbcRowSet` (**`JdbcRowSetImpl`**) para crear un objeto `JdbcRowSet`. Utilizamos la clase `JdbcRowSetImpl` aquí para demostrar la capacidad de la herramienta `JdbcRowSet`. Otras bases de datos pueden proporcionar sus propias implementaciones de `RowSet`.

En las líneas 27 a 29 se establecen las propiedades de `RowSet` que utiliza el objeto `DriverManager` para establecer una conexión a la base de datos. En la línea 27 se invoca el método **`setUrl`** de `JdbcRowSet` para especificar el URL de la base de datos. En la línea 28 se invoca el método **`setUsername`** de `JdbcRowSet` para especificar el nombre de usuario. En la línea 29 se invoca el método **`setPassword`** de `JdbcRowSet` para especificar la contraseña. En la línea 30 se invoca el método **`setCommand`** de `JdbcRowSet` para especificar la consulta SQL que se utilizará para llenar el objeto `RowSet`. En la línea 31 se invoca el método **`execute`** de `JdbcRowSet` para ejecutar la consulta SQL. El método `execute` realiza cuatro acciones: establece una conexión (`Connection`), prepara la instrucción (`Statement`) de consulta, ejecuta la consulta y almacena el objeto `ResultSet` devuelto por la consulta. Los objetos `Connection`, `Statement` y `ResultSet` se encapsulan en el objeto `JdbcRowSet`.

El resto del código es casi idéntico al de la figura 25.23, excepto que en la línea 34 se obtiene un objeto `ResultSetMetaData` del objeto `JdbcRowSet`, en la línea 44 se utiliza el método `next` de `JdbcRowSet` para obtener la siguiente fila del resultado, y en la línea 47 se utiliza el método `getObject` de `JdbcRowSet` para obtener el valor de una columna. En la línea 52 se invoca el método `close` de `JdbcRowSet`, el cual cierra los objetos `ResultSet`, `Statement` y `Connection` de `RowSet`. En un objeto `CachedRowSet`, al invocar `close` también se libera la memoria ocupada por ese objeto `RowSet`. Observe que la salida de esta aplicación es igual que la de la figura 25.23.

25.10 Java DB/Apache Derby

A partir del JDK 6, Sun Microsystems está incluyendo la base de datos basada exclusivamente en Java de código fuente abierto, llamada **Java DB** (la versión de Apache Derby producida por Sun) junto con el JDK. En la sección 25.11 utilizaremos Java DB para demostrar una nueva característica de JDBC 4.0, y demostraremos las instrucciones denominadas `PreparedStatements`. Para que pueda ejecutar la aplicación de la siguiente sección, debe configurar la base de datos `LibretaDirecciones` en Java DB. En la sección 25.11 usaremos la versión incrustada de Java DB. También hay una versión en red, que se ejecuta de manera similar al DBMS MySQL que presentamos en secciones anteriores. Para los siguientes pasos, vamos a suponer que usted está ejecutando Microsoft Windows con Java instalado en su ubicación predeterminada.

1. Java DB incluye varios archivos de procesamiento por lotes para su configuración y ejecución. Antes de ejecutar estos archivos por lotes desde un símbolo del sistema, debe establecer la variable de entorno `JAVA_HOME` para que haga referencia al directorio de instalación `C:\Archivos de programa\Java\jdk1.6.0` del JDK. Para obtener información acerca de cómo establecer el valor de una variable de entorno, consulte la sección *Antes de empezar* de este libro.
2. Abra el archivo de procesamiento por lotes `setEmbeddedCP.bat` (ubicado en `C:\Archivos de programa\Java\jdk1.6.0\db\frameworks\embedded\bin`) en un editor de texto, como el Bloc de notas. Localice la línea

   ```
   rem set DERBY_INSTALL=
   ```

   y cámbiela por

   ```
   set DERBY_INSTALL=C:\Archivos de programa\Java\jdk1.6.0\db
   ```

   Después, convierta en comentario la siguiente línea:

   ```
   @FOR %%X in ("%DERBY_HOME%") DO SET DERBY_HOME=%%~sX
   ```

   a la cual debe anteponer la palabra clave `REM`, de la siguiente manera:

   ```
   REM @FOR %%X in ("%DERBY_HOME%") DO SET DERBY_HOME=%%~sX
   ```

   Guarde sus cambios y cierre este archivo.
3. Abra una ventana de Símbolo del sistema y cambie al directorio `C:\Archivos de programa\Java\jdk1.6.0\db\frameworks\embedded\bin`. Después, escriba `setEmbeddedCP.bat` y oprima *Intro* para establecer las variables de entorno requeridas por Java DB.
4. Una base de datos Java DB incrustada debe residir en la misma ubicación que la aplicación que manipula a la base de datos. Por esta razón, cambie al directorio que contiene el código para las figuras 25.30 a 25.32. Este directorio contiene un archivo de secuencia de comandos SQL llamado `direccion.sql`, el cual crea la base de datos `LibretaDirecciones`.
5. Ejecute el comando

   ```
   "C:\Archivos de programa\Java\jdk1.6.0\db\frameworks\embedded\bin\ij"
   ```

   para iniciar la herramienta de línea de comandos para interactuar con Java DB. Las comillas dobles son necesarias, ya que la ruta contiene un espacio. Esto mostrará el indicador `ij>`.
6. En el indicador `ij>`, escriba

   ```
   connect 'jdbc:derby:LibretaDirecciones;create=true;user=jhtp7;password=jhtp7';
   ```

   para crear la base de datos `LibretaDirecciones` en el directorio actual. Este comando también crea el usuario `jhtp7` con la contraseña `jhtp7` para acceder a la base de datos.
7. Para crear la tabla de la base de datos e insertar los datos de ejemplo, escriba

   ```
   run 'direccion.sql';
   ```

8. Para terminar la herramienta de línea de comandos de Java DB, escriba

   ```
   exit;
   ```

   Ahora está listo para ejecutar la aplicación `LibretaDirecciones` en la sección 25.12.

25.11 Objetos PreparedStatement

La interfaz **`PreparedStatement`** nos permite crear instrucciones SQL compiladas, que se ejecutan con más eficiencia que los objetos `Statement`. Las instrucciones `PreparedStatement` también pueden especificar parámetros, lo cual las hace más flexibles que las instrucciones `Statement`. Los programas pueden ejecutar la misma

consulta varias veces, con distintos valores para los parámetros. Por ejemplo, en la base de datos `libros`, tal vez sea conveniente localizar todos los libros para un autor con un apellido paterno y primer nombre específicos, y ejecutar esa consulta para varios autores. Con un objeto `PreparedStatement`, esa consulta se define de la siguiente manera:

```
PreparedStatement librosAutor = connection.prepareStatement(
   "SELECT apellidoPaterno, primerNombre, titulo " +
   "FROM autores INNER JOIN isbnAutor " +
      "ON autores.idAutor=isbnAutor.idAutor " +
   "INNER JOIN titulos " +
      "ON isbnAutor.isbn=titulos.isbn " +
   "WHERE apellidoPaterno = ? AND primerNombre = ?" );
```

Los dos signos de interrogación (`?`) en la última línea de la instrucción SQL anterior son receptáculos para valores que se pasarán como parte de la consulta en la base de datos. Antes de ejecutar una instrucción `PreparedStatement`, el programa debe especificar los valores de los parámetros mediante el uso de los métodos *establecer (set)* de la interfaz `PreparedStatement`.

Para la consulta anterior, ambos parámetros son cadenas que pueden establecerse con el método **`setString`** de `PreparedStatement`, como se muestra a continuación:

```
librosAutor.setString( 1, "Deitel" );
librosAutor.setString( 2, "Paul" );
```

El primer argumento del método `setString` representa el número del parámetro que se va a establecer, y el segundo argumento es el valor de ese parámetro. Los números de los parámetros se cuentan a partir de 1, empezando con el primer signo de interrogación (`?`). Cuando el programa ejecuta la instrucción `PreparedStatement` anterior con los valores de los parámetros que se muestran aquí, la instrucción SQL que se pasa a la base de datos es

```
SELECT apellidoPaterno, primerNombre, titulo
FROM autores INNER JOIN isbnAutor
   ON autores.idAutor=isbnAutor.idAutor
INNER JOIN titulos
   ON isbnAutor.isbn=titulos.isbn
WHERE apellidoPaterno = 'Deitel' AND primerNombre = 'Paul'
```

El método `setString` escapa de manera automática los valores de los parámetros `String` según sea necesario. Por ejemplo, si el apellido paterno es O'Brien, la instrucción

```
librosAutor.setString( 1, "O'Brien" );
```

escapa el carácter `'` en O'Brien, sustituyéndolo con dos caracteres de comilla sencilla.



Tip de rendimiento 25.2

Las instrucciones `PreparedStatement` son más eficientes que las instrucciones `Statement` al ejecutar instrucciones SQL varias veces, y con distintos valores para los parámetros.



Tip para prevenir errores 25.1

Use las instrucciones `PreparedStatement` con parámetros para las consultas que reciban valores `String` como argumentos, para asegurar que los objetos `String` utilicen comillas de manera apropiada en la instrucción SQL.

La interfaz `PreparedStatement` proporciona métodos *establecer (set)* para cada tipo de SQL soportado. Es importante utilizar el método *establecer* que sea apropiado para el tipo de SQL del parámetro en la base de datos; las excepciones `SQLException` ocurren cuando un programa trata de convertir el valor de un parámetro en un tipo incorrecto. Para una lista completa de los métodos *establecer (set)* de la interfaz `PreparedStatement`, consulte la página Web `java.sun.com/javase/6/docs/api/java/sql/PreparedStatement.html`.

Aplicación "libreta de direcciones" que utiliza instrucciones **PreparedStatement**

Ahora presentaremos una aplicación "libreta de direcciones" que nos permite explorar las entradas existentes, agregar nuevas entradas y buscar entradas con un apellido paterno específico. Nuestra base de datos LibretaDirecciones de JavaDB contiene una tabla Direcciones con las columnas idDireccion, primerNombre, apellidoPaterno, email y numeroTelefonico. La columna idDireccion se denomina columna de identidad. Ésta es la manera estándar de SQL para representar una columna autoincrementada. La secuencia de comandos SQL que proporcionamos para esta base de datos utiliza la palabra clave IDENTITY de SQL para marcar la columna idDireccion como una columna de identidad. Para obtener más información acerca de la palabra IDENTITY y crear bases de datos, consulte la Guía para el desarrollador de Java DB en developers.sun.com/prodtech/javadb/reference/docs/10.2.1.6/devguide/index.html.

Nuestra aplicación de libro de direcciones consiste en tres clases: Persona (figura 25.30), ConsultasPersona (figura 25.31) y MostrarLibretaDirecciones (figura 25.32). La clase Persona es una clase simple que representa a una persona en la libreta de direcciones. Esta clase contiene campos para el ID de dirección, primer nombre, apellido paterno, dirección de e-mail y número telefónico, así como métodos *establecer* y *obtener* para manipular esos campos.

```java
// Fig. 25.30: Persona.java
// La clase Persona representa una entrada en una libreta de direcciones.
public class Persona
{
   private int idDireccion;
   private String primerNombre;
   private String apellidoPaterno;
   private String email;
   private String numeroTelefonico;

   // constructor sin argumentos
   public Persona()
   {
   } // fin del constructor de Persona sin argumentos

   // constructor
   public Persona( int id, String nombre, String apellido,
      String direccionEmail, String telefono )
   {
      establecerIDDireccion( id );
      establecerPrimerNombre( nombre );
      establecerApellidoPaterno( apellido );
      establecerEmail( direccionEmail );
      establecerNumeroTelefonico( telefono );
   } // fin del constructor de Persona con cinco argumentos

   // establece el objeto idDireccion
   public void establecerIDDireccion( int id )
   {
      idDireccion = id;
   } // fin del método establecerIDDireccion

   // devuelve el valor de idDireccion
   public int obtenerIDDireccion()
   {
      return idDireccion;
   } // fin del método obtenerIDDireccion

   // establece el primerNombre
   public void establecerPrimerNombre( String nombre )
```

Figura 25.30 | La clase Persona representa una entrada en un objeto LibretaDirecciones. (Parte 1 de 2).

```
    {
       primerNombre = nombre;
    } // fin del método establecerPrimerNombre

    // devuelve el primer nombre
    public String obtenerPrimerNombre()
    {
       return primerNombre;
    } // fin del método obtenerPrimerNombre

    // establece el apellidoPaterno
    public void establecerApellidoPaterno( String apellido )
    {
       apellidoPaterno = apellido;
    } // fin del método establecerApellidoPaterno

    // devuelve el apellido paterno
    public String obtenerApellidoPaterno()
    {
       return apellidoPaterno;
    } // fin del método obtenerApellidoPaterno

    // establece la dirección de email
    public void establecerEmail( String direccionEmail )
    {
       email = direccionEmail;
    } // fin del método establecerEmail

    // devuelve la dirección de email
    public String obtenerEmail()
    {
       return email;
    } // fin del método obtenerEmail

    // establece el número telefónico
    public void establecerNumeroTelefonico( String telefono )
    {
       numeroTelefonico = telefono;
    } // fin del método establecerNumeroTelefonico

    // devuelve el número telefónico
    public String obtenerNumeroTelefonico()
    {
       return numeroTelefonico;
    } // fin del método obtenerNumeroTelefonico
} // fin de la clase Persona
```

Figura 25.30 | La clase `Persona` representa una entrada en un objeto `LibretaDirecciones`. (Parte 2 de 2).

La clase `ConsultasPersona`

La clase `ConsultasPersona` (figura 25.31) maneja la conexión a la base de datos de la aplicación libreta de direcciones y crea las instrucciones `PreparedStatement` que utiliza la aplicación para interactuar con la base de datos. En las líneas 18 a 20 se declaran tres variables `PreparedStatement`. El constructor (líneas 23 a 49) se conecta con la base de datos en las líneas 27 y 28. Observe que no utilizamos `Class.forName` para cargar el controlador de la base de datos para Java DB, como hicimos en los ejemplos que utilizan MySQL en secciones anteriores de este capítulo. JDBC 4.0, que forma parte de Java SE 6, soporta el **descubrimiento automático de controladores**; ya no tenemos que cargar el controlador de la base de datos por adelantado. Al momento de escribir este libro, esta característica se encuentra en proceso de implementarse en MySQL.

```
// Fig. 25.31: ConsultasPersona.java
// Instrucciones PreparedStatement utilizadas por la aplicación Libreta de direcciones
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.List;
import java.util.ArrayList;

public class ConsultasPersona
{
   private static final String URL = "jdbc:derby:LibretaDirecciones";
   private static final String NOMBREUSUARIO = "jhtp7";
   private static final String CONTRASENIA = "jhtp7";

   private Connection conexion = null; // maneja la conexión
   private PreparedStatement seleccionarTodasLasPersonas = null;
   private PreparedStatement seleccionarPersonasPorApellido = null;
   private PreparedStatement insertarNuevaPersona = null;

   // constructor
   public ConsultasPersona()
   {
      try
      {
         conexion =
            DriverManager.getConnection( URL, NOMBREUSUARIO, CONTRASENIA );

         // crea una consulta que selecciona todas las entradas en la LibretaDirecciones
         seleccionarTodasLasPersonas =
            conexion.prepareStatement( "SELECT * FROM Direcciones" );

         // crea una consulta que selecciona las entradas con un apellido específico
         seleccionarPersonasPorApellido = conexion.prepareStatement(
            "SELECT * FROM Direcciones WHERE ApellidoPaterno = ?" );

         // crea instrucción insert para agregar una nueva entrada en la base de 39
         datos
         insertarNuevaPersona = conexion.prepareStatement(
            "INSERT INTO Direcciones " +
            "( PrimerNombre, ApellidoPaterno, Email, NumeroTelefonico ) " +
            "VALUES ( ?, ?, ?, ? )" );
      } // fin de try
      catch ( SQLException excepcionSql )
      {
         excepcionSql.printStackTrace();
         System.exit( 1 );
      } // fin de catch
   } // fin del constructor de ConsultasPersona

   // selecciona todas las direcciones en la base de datos
   public List< Persona > obtenerTodasLasPersonas()
   {
      List< Persona > resultados = null;
      ResultSet conjuntoResultados = null;

      try
```

Figura 25.31 | Una interfaz que almacena todas las consultas para que las utilice un objeto `LibretaDirecciones`. (Parte 1 de 4).

```
      {
         // executeQuery devuelve ResultSet que contiene las entradas que coinciden
         conjuntoResultados = seleccionarTodasLasPersonas.executeQuery();
         resultados = new ArrayList< Persona >();

         while ( conjuntoResultados.next() )
         {
            resultados.add( new Persona(
              conjuntoResultados.getInt( "idDireccion" ),
              conjuntoResultados.getString( "primerNombre" ),
              conjuntoResultados.getString( "apellidoPaterno" ),
              conjuntoResultados.getString( "email" ),
              conjuntoResultados.getString( "numeroTelefonico" ) ) );
         } // fin de while
      } // fin de try
      catch ( SQLException excepcionSql )
      {
         excepcionSql.printStackTrace();
      } // fin de catch
      finally
      {
         try
         {
            conjuntoResultados.close();
         } // fin de try
         catch ( SQLException excepcionSql )
         {
            excepcionSql.printStackTrace();
            close();
         } // fin de catch
      } // fin de finally

      return resultados;
   } // fin del método obtenerTodasLasPersonaas

   // selecciona persona por apellido paterno

   public List< Persona > obtenerPersonasPorApellido( String nombre )
   {
      List< Persona > resultados = null;
      ResultSet conjuntoResultados = null;

      try
      {
         seleccionarPersonasPorApellido.setString( 1, nombre ); // especifica el apellido paterno

         // executeQuery devuelve ResultSet que contiene las entradas que coinciden
         conjuntoResultados = seleccionarPersonasPorApellido.executeQuery();

         resultados = new ArrayList< Persona >();

         while ( conjuntoResultados.next() )
         {
            resultados.add( new Persona(
              conjuntoResultados.getInt( "idDireccion" ),
              conjuntoResultados.getString( “primerNombre" ),
              conjuntoResultados.getString( "apellidoPaterno" ),
```

Figura 25.31 | Una interfaz que almacena todas las consultas para que las utilice un objeto LibretaDirecciones. (Parte 2 de 4).

```
                 conjuntoResultados.getString( "email" ),
                 conjuntoResultados.getString( "numeroTelefonico" ) ) );
         } // fin de while
      } // fin de try
      catch ( SQLException excepcionSql )
      {
         excepcionSql.printStackTrace();
      } // fin de catch
      finally
      {
         try
         {
            conjuntoResultados.close();
         } // fin de try
         catch ( SQLException excepcionSql )
         {
            excepcionSql.printStackTrace();
            close();
         } // fin de catch
      } // fin de finally

      return resultados;
   } // fin del método obtenerPersonasPorApellido

   // agrega una entrada
   public int agregarPersona(
      String pnombre, String apaterno, String email, String num )
   {
      int resultado = 0;

      // establece los parámetros, después ejecuta insertarNuevaPersona
      try
      {
         insertarNuevaPersona.setString( 1, pnombre );
         insertarNuevaPersona.setString( 2, apaterno );
         insertarNuevaPersona.setString( 3, email );
         insertarNuevaPersona.setString( 4, num );

         // inserta la nueva entrada; devuelve # de filas actualizadas
         resultado = insertarNuevaPersona.executeUpdate();
      } // fin de try
      catch ( SQLException excepcionSql )
      {
         excepcionSql.printStackTrace();
         close();
      } // fin de catch

      return resultado;
   } // fin del método agregarPersona

   // cierra la conexión a la base de datos
   public void close()
   {
      try
      {
         conexion.close();
      } // fin de try
      catch ( SQLException excepcionSql )
```

Figura 25.31 | Una interfaz que almacena todas las consultas para que las utilice un objeto `LibretaDirecciones`. (Parte 3 de 4).

```
173         {
174            excepcionSql.printStackTrace();
175         } // fin de catch
176      } // fin del método close
177   } // fin de la interfaz ConsultasPersona
```

Figura 25.31 | Una interfaz que almacena todas las consultas para que las utilice un objeto LibretaDirecciones. (Parte 4 de 4).

En las líneas 31 y 32 se invoca el método **prepareStatement** de Connection para crear la instrucción PreparedStatement llamada seleccionarTodasLasPersonas, la cual selecciona todas las filas en la tabla Direcciones. En las líneas 35 y 36 se crea la instrucción PreparedStatement llamada seleccionarPersonasPorApellido con un parámetro. Esta instrucción selecciona todas las filas en la tabla Direcciones que coincidan con un apellido específico. Observe el carácter ? que se utiliza para especificar el parámetro apellido. En las líneas 39 a 42 se crea la instrucción PreparedStatement llamada insertarNuevaPersona, concuatro parámetros que representan el primer nombre, apellido paterno, dirección de e-mail y número telefónico para una nueva entrada. De nuevo, observe los caracteres ? que se utilizan para representar estos parámetros.

El método obtenerTodasLasPersonas (líneas 52 a 91) ejecuta la instrucción PreparedStatement seleccionarTodasLasPersonas (línea 60) mediante una llamada al método **executeQuery**, el cual devuelve un objeto ResultSet que contiene las filas que coinciden con la consulta (en este caso, todas las filas en la tabla Direcciones). En las líneas 61 a 71 se colocan los resultados de la consulta en un objeto ArrayList de objetos Persona, el cual se devuelve en la línea 90 al método que hizo la llamada. El método obtenerPersonasPorApellido (líneas 95 a 137) utiliza el método **setString** de PreparedStatement para establecer el parámetro en seleccionarPersonasPorApellido. Después, en la línea 105 se ejecuta la consulta y en las líneas 107 a 117 se colocan los resultados de la consulta en un objeto ArrayList de objetos Persona. En la línea 136 se devuelve el objeto ArrayList al método que hizo la llamada.

El método agregarPersona (líneas 140 a 163) utiliza el método setString de PreparedStatement (líneas 148 a 151) para establecer los parámetros para la instrucción PreparedStatement llamada insertarNuevaPersona. La línea 154 utiliza el método **executeUpdate** de PreparedStatement para insertar un nuevo registro. Este método devuelve un entero, el cual indica el número de filas que se actualizaron (o insertaron) en la base de datos. El método close (líneas 166 a 176) simplemente cierra la conexión a la base de datos.

La clase **MostrarLibretaDirecciones**

La aplicación MostrarLibretaDirecciones (figura 25.32) utiliza un objeto de la clase ConsultasPersona para interactuar con la base de datos. En la línea 59 se crea el objeto ConsultasPersona que se utiliza en la clase MostrarLibretaDirecciones. Cuando el usuario oprime el objeto JButton llamado **Explorar todas las entradas**, se hace una llamada al manejador botonExplorarActionPerformed (líneas 309 a 335). En la línea 313 se hace una llamada al método obtenerTodasLasPersonas en el objeto ConsultasPersona para obtener todas las entradas en la base de datos. Después, el usuario puede desplazarse a través de las entradas, usando los objetos JButton **Anterior** y **Siguiente**. Cuando el usuario oprime el objeto JButton **Buscar**, se hace una llamada al manejador botonConsultaActionPerformed (líneas 265 a 287). En las líneas 267 y 268 se hace una llamada al método obtenerPersonasPorApellido en el objeto ConsultasPersona, para obtener las entradas en la base de datos que coincidan con el apellido paterno especificado. Si hay varias entradas de este tipo, el usuario puede desplazarse de una entrada a otra mediante los objetos JButton **Anterior** y **Siguiente**.

Para agregar una nueva entrada a la base de datos LibretaDirecciones, el usuario puede escribir el primer nombre, apellido paterno, email y número telefónico (el valor de IdDireccion se autoincrementará) en los objetos JTextField y oprimir el objeto JButton **Insertar nueva entrada**. Cuando el usuario oprime este botón, se hace una llamada al manejador botonInsertarActionPerformed (líneas 338 a 352). En las líneas 340 a 342 se hace una llamada al método agregarPersona en el objeto ConsultasPersona, para agregar una nueva entrada a la base de datos.

Después, el usuario puede ver distintas entradas oprimiendo los objetos JButton **Anterior** o **Siguiente**, lo cual produce llamadas a los métodos botonAnteriorActionPerformed (líneas 241 a 250) o botonSiguien-

```java
// Fig. 25.32: MostrarLibretaDirecciones.java
// Una libreta de direcciones simple
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
import java.awt.FlowLayout;
import java.awt.GridLayout;
import java.util.List;
import javax.swing.JButton;
import javax.swing.Box;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JTextField;
import javax.swing.WindowConstants;
import javax.swing.BoxLayout;
import javax.swing.BorderFactory;
import javax.swing.JOptionPane;

public class MostrarLibretaDirecciones extends JFrame
{
   private Persona entradaActual;
   private ConsultasPersona consultasPersona;
   private List< Persona > resultados;
   private int numeroDeEntradas = 0;
   private int indiceEntradaActual;

   private JButton botonExplorar;
   private JLabel etiquetaEmail;
   private JTextField campoTextoEmail;
   private JLabel etiquetaPrimerNombre;
   private JTextField campoTextoPrimerNombre;
   private JLabel etiquetaID;
   private JTextField campoTextoID;
   private JTextField campoTextoIndice;
   private JLabel etiquetaApellidoPaterno;
   private JTextField campoTextoApellidoPaterno;
   private JTextField campoTextoMax;
   private JButton botonSiguiente;
   private JLabel etiquetaDe;
   private JLabel etiquetaTelefono;
   private JTextField campoTextoTelefono;
   private JButton botonAnterior;
   private JButton botonConsulta;
   private JLabel etiquetaConsulta;
   private JPanel panelConsulta;
   private JPanel panelNavegar;
   private JPanel panelMostrar;
   private JTextField campoTextoConsulta;
   private JButton botonInsertar;

   // constructor sin argumentos
   public MostrarLibretaDirecciones()
   {
      super( "Libreta de direcciones" );

      // establece la conexión a la base de datos y las instrucciones PreparedStatement
      consultasPersona = new ConsultasPersona();
```

Figura 25.32 | Una libreta de direcciones simple. (Parte 1 de 7).

```

      // crea la GUI
      panelNavegar = new JPanel();
      botonAnterior = new JButton();
      campoTextoIndice = new JTextField( 2 );
      etiquetaDe = new JLabel();
      campoTextoMax = new JTextField( 2 );
      botonSiguiente = new JButton();
      panelMostrar = new JPanel();
      etiquetaID = new JLabel();
      campoTextoID = new JTextField( 10 );
      etiquetaPrimerNombre = new JLabel();
      campoTextoPrimerNombre = new JTextField( 10 );
      etiquetaApellidoPaterno = new JLabel();
      campoTextoApellidoPaterno = new JTextField( 10 );
      etiquetaEmail = new JLabel();
      campoTextoEmail = new JTextField( 10 );
      etiquetaTelefono = new JLabel();
      campoTextoTelefono = new JTextField( 10 );
      panelConsulta = new JPanel();
      etiquetaConsulta = new JLabel();
      campoTextoConsulta = new JTextField( 10 );
      botonConsulta = new JButton();
      botonExplorar = new JButton();
      botonInsertar = new JButton();

      setLayout( new FlowLayout( FlowLayout.CENTER, 10, 10 ) );
      setSize( 400, 300 );
      setResizable( false );

      panelNavegar.setLayout(
         new BoxLayout( panelNavegar, BoxLayout.X_AXIS ) );

      botonAnterior.setText( "Anterior" );
      botonAnterior.setEnabled( false );
      botonAnterior.addActionListener(
         new ActionListener()
         {
            public void actionPerformed( ActionEvent evt )
            {
               botonAnteriorActionPerformed( evt );
            } // fin del método actionPerformed
         } // fin de la clase interna anónima
      ); // fin de la llamada a addActionListener

      panelNavegar.add( botonAnterior );
      panelNavegar.add( Box.createHorizontalStrut( 10 ) );

      campoTextoIndice.setHorizontalAlignment(
         JTextField.CENTER );
      campoTextoIndice.addActionListener(
         new ActionListener()
         {
            public void actionPerformed( ActionEvent evt )
            {
               campoTextoIndiceActionPerformed( evt );
            } // fin del método actionPerformed
         } // fin de la clase interna anónima
      ); // fin de la llamada a addActionListener
```

Figura 25.32 | Una libreta de direcciones simple. (Parte 2 de 7).

```
         panelNavegar.add( campoTextoIndice );
         panelNavegar.add( Box.createHorizontalStrut( 10 ) );

         etiquetaDe.setText( "de" );
         panelNavegar.add( etiquetaDe );
         panelNavegar.add( Box.createHorizontalStrut( 10 ) );

         campoTextoMax.setHorizontalAlignment(
            JTextField.CENTER );
         campoTextoMax.setEditable( false );
         panelNavegar.add( campoTextoMax );
         panelNavegar.add( Box.createHorizontalStrut( 10 ) );

         botonSiguiente.setText( "Siguiente" );
         botonSiguiente.setEnabled( false );
         botonSiguiente.addActionListener(
            new ActionListener()
            {
               public void actionPerformed( ActionEvent evt )
               {
                  botonSiguienteActionPerformed( evt );
               } // fin del método actionPerformed
            } // fin de la clase interna anónima
         ); // fin de la llamada a addActionListener

         panelNavegar.add( botonSiguiente );
         add( panelNavegar );

         panelMostrar.setLayout( new GridLayout( 5, 2, 4, 4 ) );

         etiquetaID.setText( "ID Direccion:" );
         panelMostrar.add( etiquetaID );

         campoTextoID.setEditable( false );
         panelMostrar.add( campoTextoID );

         etiquetaPrimerNombre.setText( "Primer nombre:" );
         panelMostrar.add( etiquetaPrimerNombre );
         panelMostrar.add( campoTextoPrimerNombre );

         etiquetaApellidoPaterno.setText( "Apellido paterno:" );
         panelMostrar.add( etiquetaApellidoPaterno );
         panelMostrar.add( campoTextoApellidoPaterno );

         etiquetaEmail.setText( "Email:" );
         panelMostrar.add( etiquetaEmail );
         panelMostrar.add( campoTextoEmail );

         etiquetaTelefono.setText( "Telefono:" );
         panelMostrar.add( etiquetaTelefono );
         panelMostrar.add( campoTextoTelefono );
         add( panelMostrar );

         panelConsulta.setLayout(
            new BoxLayout( panelConsulta, BoxLayout.X_AXIS) );

         panelConsulta.setBorder( BorderFactory.createTitledBorder(
            "Buscar una entrada por apellido" ) );
```

Figura 25.32 | Una libreta de direcciones simple. (Parte 3 de 7).

```
      etiquetaConsulta.setText( "Apellido paterno:" );
      panelConsulta.add( Box.createHorizontalStrut( 5 ) );
      panelConsulta.add( etiquetaConsulta );
      anelConsulta.add( Box.createHorizontalStrut( 10 ) );
      panelConsulta.add( campoTextoConsulta );
      panelConsulta.add( Box.createHorizontalStrut( 10 ) );

      botonConsulta.setText( "Buscar" );
      botonConsulta.addActionListener(
         new ActionListener()
         {
            public void actionPerformed( ActionEvent evt )
            {
              botonConsultaActionPerformed( evt );
            } // fin del método actionPerformed
         } // fin de la clase interna anónima
      ); // fin de la llamada a addActionListener

      panelConsulta.add( botonConsulta );
      panelConsulta.add( Box.createHorizontalStrut( 5 ) );
      add( panelConsulta );

      botonExplorar.setText( "Explorar todas las entradas" );
      botonExplorar.addActionListener(
         new ActionListener()
         {
            public void actionPerformed( ActionEvent evt )
            {
              botonExplorarActionPerformed( evt );
            } // fin del método actionPerformed
         } // fin de la clase interna anónima
      ); // fin de la llamada a addActionListener

      add( botonExplorar );

      botonInsertar.setText( "Insertar nueva entrada" );
      botonInsertar.addActionListener(
         new ActionListener()
         {
            public void actionPerformed( ActionEvent evt )
            {
              botonInsertarActionPerformed( evt );
            } // fin del método actionPerformed
         } // fin de la clase interna anónima
      ); // fin de la llamada a addActionListener

      add( botonInsertar );

       addWindowListener(
         new WindowAdapter()
         {
            public void windowClosing( WindowEvent evt )
            {
              consultasPersona.close(); // cierra la conexión a la base de datos
              System.exit( 0 );
            } // fin del método windowClosing
         } // fin de la clase interna anónima
      ); // fin de la llamada a addWindowListener

```

Figura 25.32 | Una libreta de direcciones simple. (Parte 4 de 7).

```
      setVisible( true );
   } // fin del constructor sin argumentos

   // maneja la llamada cuando se hace clic en botonAnterior
   private void botonAnteriorActionPerformed( ActionEvent evt )
   {
      indiceEntradaActual--;

      if ( indiceEntradaActual < 0 )
         indiceEntradaActual = numeroDeEntradas - 1;

      campoTextoIndice.setText( "" + ( indiceEntradaActual + 1 ) );
      campoTextoIndiceActionPerformed( evt );
   } // fin del método botonAnteriorActionPerformed

   // maneja la llamada cuando se hace clic en botonSiguiente
   private void botonSiguienteActionPerformed( ActionEvent evt )
   {
      indiceEntradaActual++;

      if ( indiceEntradaActual >= numeroDeEntradas )
         indiceEntradaActual = 0;

      campoTextoIndice.setText( "" + ( indiceEntradaActual + 1 ) );
      campoTextoIndiceActionPerformed( evt );
   } // fin del método botonSiguienteActionPerformed

   // maneja la llamada cuando se hace clic en botonConsulta
   private void botonConsultaActionPerformed( ActionEvent evt )
   {
      resultados =
         consultasPersona.obtenerPersonasPorApellido( campoTextoConsulta.getText() );
      numeroDeEntradas = resultados.size();

      if ( numeroDeEntradas != 0 )
      {
         indiceEntradaActual = 0;
         entradaActual = resultados.get( indiceEntradaActual );
         campoTextoID.setText( "" + entradaActual.obtenerIDDireccion() );
         campoTextoPrimerNombre.setText( entradaActual.obtenerPrimerNombre() );
         campoTextoApellidoPaterno.setText( entradaActual.obtenerApellidoPaterno() );
         campoTextoEmail.setText( entradaActual.obtenerEmail() );
         campoTextoTelefono.setText( entradaActual.obtenerNumeroTelefonico() );
         campoTextoMax.setText( "" + numeroDeEntradas );
         campoTextoIndice.setText( "" + ( indiceEntradaActual + 1 ) );
         botonSiguiente.setEnabled( true );
         botonAnterior.setEnabled( true );
      } // fin de if
      else
         botonExplorarActionPerformed( evt );
   } // fin del método botonConsultaActionPerformed

   // maneja la llamada cuando se introduce un nuevo valor en campoTextoIndice
   private void campoTextoIndiceActionPerformed( ActionEvent evt )
   {
      indiceEntradaActual =
         ( Integer.parseInt( campoTextoIndice.getText() ) - 1 );

      if ( numeroDeEntradas != 0 && indiceEntradaActual < numeroDeEntradas )
```

Figura 25.32 | Una libreta de direcciones simple. (Parte 5 de 7).

```
         {
            entradaActual = resultados.get( indiceEntradaActual );
            campoTextoID.setText("" + entradaActual.obtenerIDDireccion() );
            campoTextoPrimerNombre.setText( entradaActual.obtenerPrimerNombre() );
            campoTextoApellidoPaterno.setText( entradaActual.obtenerApellidoPaterno() );
            campoTextoEmail.setText( entradaActual.obtenerEmail() );
            campoTextoTelefono.setText( entradaActual.obtenerNumeroTelefonico() );
            campoTextoMax.setText( "" + numeroDeEntradas );
            campoTextoIndice.setText( "" + ( indiceEntradaActual + 1 ) );
         } // fin de if
      } // fin del método campoTextoIndiceActionPerformed

      // maneja la llamada cuando se hace clic en botonExplorar
      private void botonExplorarActionPerformed( ActionEvent evt )
      {
         try
         {
            resultados = consultasPersona.obtenerTodasLasPersonas();
            numeroDeEntradas = resultados.size();

            if ( numeroDeEntradas != 0 )
            {
               indiceEntradaActual = 0;
               entradaActual = resultados.get( indiceEntradaActual );
               campoTextoID.setText( "" + entradaActual.obtenerIDDireccion() );
               campoTextoPrimerNombre.setText( entradaActual.obtenerPrimerNombre() );
               campoTextoApellidoPaterno.setText( entradaActual.obtenerApellidoPaterno() );
               campoTextoEmail.setText( entradaActual.obtenerEmail() );
               campoTextoTelefono.setText( entradaActual.obtenerNumeroTelefonico() );
               campoTextoMax.setText( "" + numeroDeEntradas );
               campoTextoIndice.setText( "" + ( indiceEntradaActual + 1 ) );
               botonSiguiente.setEnabled( true );
               botonAnterior.setEnabled( true );
            } // fin de if
         } // fin de try
         catch ( Exception e )
         {
            e.printStackTrace();
         } // fin de catch
      } // fin del método botonExplorarActionPerformed

      // maneja la llamada cuando se hace clic en botonInsertar
      private void botonInsertarActionPerformed( ActionEvent evt )
      {
         int resultado = consultasPersona.agregarPersona( campoTextoPrimerNombre.getText(),
            campoTextoApellidoPaterno.getText(), campoTextoEmail.getText(),
            campoTextoTelefono.getText() );

         if ( resultado == 1 )
            JOptionPane.showMessageDialog( this, "Se agrego persona!",
               "Se agrego persona", JOptionPane.PLAIN_MESSAGE );
         else
            JOptionPane.showMessageDialog( this, "No se agrego persona!",
               "Error", JOptionPane.PLAIN_MESSAGE );

         botonExplorarActionPerformed( evt );
      } // fin del método botonInsertarActionPerformed

      // método main
```

Figura 25.32 | Una libreta de direcciones simple. (Parte 6 de 7).

```
355      public static void main( String args[] )
356      {
357         new MostrarLibretaDirecciones();
358      } // fin del método main
359   } // fin de la clase MostrarLibretaDireccione
```

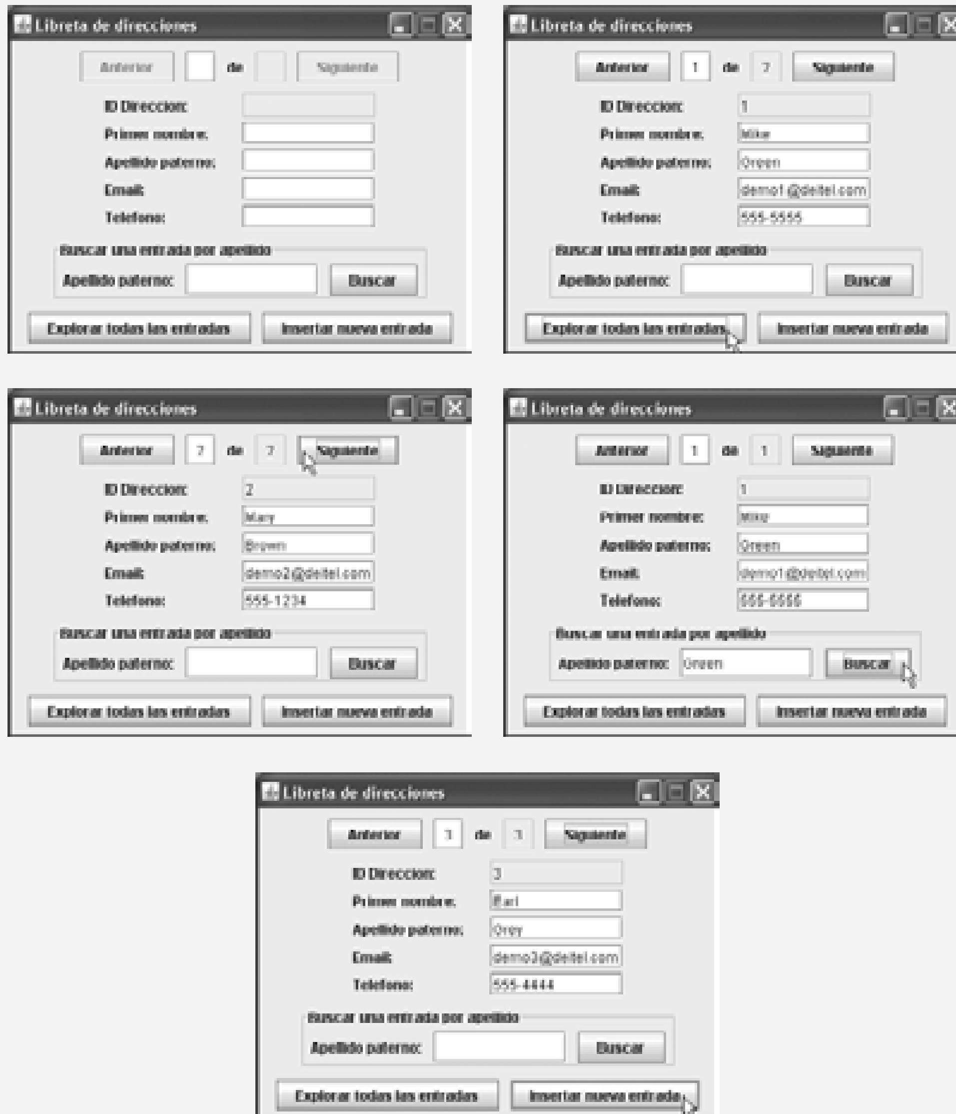


Figura 25.32 | Una libreta de direcciones simple. (Parte 7 de 7).

`teActionPerformed` (líneas 253 a 262), respectivamente. De manera alternativa, el usuario puede escribir un número en el objeto `campoTextoIndice` y oprimir *Intro* para ver una entrada específica.

25.12 Procedimientos almacenados

Muchos sistemas de administración de bases de datos pueden almacenar instrucciones de SQL individuales o conjuntos de instrucciones de SQL en una base de datos, para que los programas que tengan acceso a esa base de datos puedan invocar esas instrucciones. A dichas instrucciones de SQL se les conoce como **procedimientos**

almacenados. JDBC permite a los programas invocar procedimientos almacenados mediante el uso de objetos que implementen a la interfaz **`CallableStatement`**. Los objetos `CallableStatement` pueden recibir argumentos especificados con los métodos heredados de la interfaz `PreparedStatement`. Además, los objetos `CallableStatement` pueden especificar **parámetros de salida**, en los cuales un procedimiento almacenado puede colocar valores de retorno. La interfaz `CallableStatement` incluye métodos para especificar cuáles parámetros en un procedimiento almacenado son parámetros de salida. La interfaz también incluye métodos para obtener los valores de los parámetros de salida devueltos de un procedimiento almacenado.



Tip de portabilidad 25.6

Aunque la sintaxis para crear procedimientos almacenados difiere entre los diversos sistemas de administración de bases de datos, la interfaz `CallableStatement` *proporciona una interfaz uniforme para especificar los parámetros de entrada y salida de los procedimientos almacenados, así como para invocarlos.*



Tip de portabilidad 25.7

De acuerdo con la documentación de la API para la interfaz `CallableStatement`*, para lograr una máxima portabilidad entre los sistemas de bases de datos, los programas deben procesar las cuentas de actualización o los objetos* `ResultSet` *devueltos de un objeto* `CallableStatement` *antes de obtener los valores de cualquier parámetro de salida.*

25.13 Procesamiento de transacciones

Muchas aplicaciones de bases de datos requieren garantías de que una serie de operaciones de inserción, actualización y eliminación se ejecuten de manera apropiada, antes de que la aplicación continúe procesando la siguiente operación en la base de datos. Por ejemplo, al transferir dinero por medios electrónicos entre dos cuentas de banco, varios factores determinan si la transacción fue exitosa. Empezamos por especificar la cuenta de origen y el monto que deseamos transferir de esa cuenta hacia una cuenta de destino. Después, especificamos la cuenta de destino. El banco comprueba la cuenta de origen para determinar si hay suficientes fondos en ella como para poder completar la transferencia. De ser así, el banco retira el monto especificado de la cuenta de origen y, si todo sale bien, deposita el dinero en la cuenta de destino para completar la transferencia. ¿Qué ocurre si la transferencia falla después de que el banco retira el dinero de la cuenta de origen? En un sistema bancario apropiado, el banco vuelve a depositar el dinero en la cuenta de origen. ¿Cómo se sentiría usted si el dinero se restara de su cuenta de origen y el banco *no* depositara el dinero en la cuenta de destino?

El **procesamiento de transacciones** permite a un programa que interactúa con una base de datos tratar una operación en la base de datos (o un conjunto de operaciones) como una sola operación. Dicha operación también se conoce como **operación atómica** o **transacción**. Al final de una transacción, se puede tomar una de dos decisiones: **confirmar (commit) la transacción** o **rechazar (roll back) la transacción**. Al confirmar la transacción se finaliza(n) la(s) operación(es) de la base de datos; todas las inserciones, actualizaciones y eliminaciones que se llevaron a cabo como parte de la transacción no pueden invertirse sin tener que realizar una nueva operación en la base de datos. Al rechazar la transacción, la base de datos queda en el estado anterior a la operación. Esto es útil cuando una parte de una transacción no se completa en forma apropiada. En nuestra discusión acerca de la transferencia entre cuentas bancarias, la transacción se rechazaría si el depósito no pudiera realizarse en la cuenta de destino.

Java ofrece el procesamiento de transacciones a través de varios métodos de la interfaz `Connection`. El método **`setAutoCommit`** especifica si cada instrucción SQL se confirma una vez completada (un argumento `true`), o si deben agruparse varias instrucciones SQL para formar una transacción (un argumento `false`). Si el argumento para `setAutoCommit` es `false`, el programa debe colocar después de la última instrucción SQL en la transacción una llamada al método **`commit`** de `Connection` (para confirmar los cambios en la base de datos) o al método **`rollback`** de `Connection` (para regresar la base de datos al estado anterior a la transacción). La interfaz `Connection` también cuenta con el método **`getAutoCommit`** para determinar el estado de autoconfirmación para el objeto `Connection`.

25.14 Conclusión

En este capítulo aprendió acerca de los conceptos básicos de las bases de datos, cómo interactuar con los datos en una base de datos mediante el uso de SQL y cómo usar JDBC para permitir que las aplicaciones de Java manipu-

len bases de datos MySQL y Java DB. Aprendió acerca de los comandos `SELECT`, `INSERT`, `UPDATE` y `DELETE` de SQL, y también acerca de las cláusulas como `WHERE`, `ORDER BY` e `INNER JOIN`. Conoció los pasos explícitos para obtener una conexión (`Connection`) a la base de datos, crear una instrucción (`Statement`) para interactuar con los datos de la base de datos, ejecutar la instrucción y procesar los resultados. Después, vio cómo usar un objeto `RowSet` para simplificar el proceso de conectarse a una base de datos y crear instrucciones. Utilizó instrucciones `PreparedStatement` para crear instrucciones de SQL precompiladas. Aprendió también a crear y configurar bases de datos, tanto en MySQL como en Java DB. También vimos las generalidades acerca de las instrucciones `CallableStatement` y el procesamiento de transacciones. En el siguiente capítulo, aprenderá acerca del desarrollo de aplicaciones Web con JavaServer Faces. Las aplicaciones basadas en Web crean contenido que, por lo general, se muestra en los clientes exploradores Web. Como veremos en el capítulo 27, las aplicaciones Web también pueden usar la API JDBC para acceder a las bases de datos y crear contenido Web más dinámico.

25.15 Recursos Web y lecturas recomendadas

`java.sun.com/products/jdbc`
La página inicial sobre JDBC de Sun Microsystems, Inc.

`java.sun.com/docs/books/tutorial/jdbc/index.html`
La trayectoria del *Tutorial de Java sobre JDBC*.

`industry.java.sun.com/products/jdbc/drivers`
El motor de búsqueda de Sun Microsystems para localizar controladores de JDBC.

`www.sql.org`
Este portal de SQL proporciona vínculos hacia muchos recursos, incluyendo la sintaxis de SQL, tips, tutoriales, libros, revistas, grupos de discusión, compañías con servicios de SQL, consultores de SQL y software gratuito.

`www.datadirect.com/developer/jdbc/topics/perfoptjdbc/index.ssp`
Informe que describe cómo diseñar una buena aplicación de JDBC.

`java.sun.com/javase/6/docs/technotes/guides/jdbc/index.html`
La documentación de la API JDBC de Sun Microsystems, Inc.

`java.sun.com/products/jdbc/faq.html`
Las preguntas frecuentes acerca de JDBC de Sun Microsystems, Inc.

`www.jguru.com/faq/JDBC`
Las FAQs sobre JDBC de JGuru.

`www.mysql.com`
Este sitio es la página inicial de la base de datos MySQL. Puede descargar las versiones más recientes de MySQL y MySQL Connector/J, y acceder a su documentación en línea.

`www.mysql.com/products/enterprise/server.html`
Introducción al servidor de bases de datos MySQL y vínculos a sus sitios de documentación y descargas.

`dev.mysql.com/doc/mysql/en/index.html`
`dev.mysql.com/doc/refman/5.0/es/index.html`
Manual de referencia de MySQL, en inglés y en español.

`dev.mysql.com/doc/refman/5.1/en/connector-mxj.html`
Documentación sobre MySQL Connector/J, incluyendo instrucciones de instalación y ejemplos.

`developers.sun.com/prodtech/javadb/reference/docs/10.2.1.6/devguide/index.html`
La *Guía para el desarrollador de Java DB*.

`java.sun.com/javase/6/docs/technotes/guides/jdbc/getstart/rowsetImpl.html`
Presenta las generalidades acerca de la interfaz `RowSet` y sus subinterfaces. Este sitio también habla sobre las implementaciones de referencia de estas interfaces de Sun y su uso.

`developer.java.sun.com/developer/Books/JDBCTutorial/chapter5.html`
El capítulo 5 (tutorial de `RowSet`) del libro *The JDBC 2.0 API Tutorial and Reference, Segunda edición*.

Lecturas recomendadas

Ashmore, D. C., “Best Practices for JDBC Programming”, *Java Developers Journal*, 5: núm. 4 (2000), 42 a 54.

Blaha, M. R., W. J. Premerlani y J. E. Rumbaugh, “Relational Database Design Using an Object-Oriented Methodology”, *Communications of the ACM*, 31: núm. 4 (1988): 414 a 427.

Brunner, R. J., “The Evolution of Connecting”, *Java Developers Journal*, 5: núm. 10 (2000): 24 a 26.

Brunner, R. J., "After the Connection", *Java Developers Journal*, 5: núm. 11 (2000): 42 a 46.
Callahan, T., "So You Want a Stand-Alone Database for Java", *Java Developers Journal*, 3: núm. 12 (1998): 28 a 36.
Codd, E. F. "A Relational Model of Data for Large Shared Data Banks", *Communications of the ACM*, Junio 1970.
Codd, E. F. "Further Normalization of the Data Base Relational Model", *Courant Computer Science Symposia*, Vol. 6, *Data Base Systems*. Upple Saddle River, NJ: Prentice Hall, 1972.
Codd, E. F. "Fatal Flaws in SQL". *Datamation*, 34: núm. 16 (1988): 45 a 48.
Cooper, J. W. "Making Databases Easier for Your Users", *Java Pro*, 4: núm. 10 (2000): 47 a 54.
Date, C. J. *An Introduction to Database Systems, 8/e*. Reading, MA: Pearson Education, 2003.
Deitel, H. M., P. J. Deitel y D. R. Choffnes. *Operating Systems, Tercera edición*. Upper Saddle River, NJ: Prentice Hall, 2004.
Duguay, C. "Electronic Mail Merge". *Java Pro*, Invierno 1999/2000, 22 a 32.
Ergul, S. "Transaction Processing with Java". *Java Report*, Enero 2001, 30 a 36.
Fisher, M. "JDBC Database Access" (una trayectoria en *The Java Tutorial*), <java.sun.com/docs/books/tutorial/jdbc/index.html>.
Harrison, G., "Browsing the JDBC API". *Java Developers Journal*, 3: núm. 2 (1998): 44 a 52.
Jasnowski, M., "persistente Frameworks". *Java Developers Journal*, 5: núm. 11 (2000): 82 a 86.
"JDBC API Documentation". <java.sun.com/javase/6/docs/technotes/guides/jdbc/>.
Jordan, D. "An Overview of Sun's Java Data Objects Specification". *Java Pro*, 4: núm. 6 (2000): 102 a 108.
Khanna, P. "Managing Object Persistente with JDBC". *Java Pro*, 4: núm. 5 (2000): 28 a 33.
Reese, G. *Database Programming with JDBC and Java, Segunda edición*. Cambridge, MA: O'Reilly, 2001.
Spell, B. "Create Enterprise Applications with JDBC 2.0". *Java Pro,* 4: núm. 4 (2000): 40 a 44.
Stonebraker, M. "Operating System Support for Database Management". *Communications of the ACM*, 24: núm. 7 (1981): 412 a 418.
Taylor, A. *JDBC Developer's Resource: Database Programming on the Internet*. Upper Saddle River, NJ: Prentice Hall, 1999.
Thilmany, C. "Applying Patterns to JDBC Development". *Java Developers Journal*, 5: núm. 6 (2000): 80 a 90.
Venugopal, S. 2000. "Cross-Database Portability with JDBC". *Java Developers Journal,* 5: núm. 1 (2000): 58-62.
White, S., M. Fisher, R. Cattell, G. Hamilton y M. Hapner. *JDBC API Tutorial and Reference, Segunda edición*. Boston, MA: Addison Wesley, 1999.
Winston, A. "A Distributed Database Primer". *UNIX World*, Abril 1988, 54 a 63.

Resumen

Sección 25.1 Introducción

- Una base de datos es una colección integrada de datos. Un sistema de administración de bases de datos (DBMS) proporciona mecanismos para almacenar, organizar, obtener y modificar datos para muchos usuarios.
- Los sistemas de administración de bases de datos más populares hoy en día son los sistemas de bases de datos relacionales.
- SQL es el lenguaje estándar internacional, utilizado casi universalmente con sistemas de bases de datos relacionales, para realizar consultas y manipular datos.
- Los programas en Java se conectan a (e interactúan con) bases de datos relacionales a través de una interfaz: software que facilita las comunicaciones entre un sistema de administración de bases de datos y un programa.
- Los programas en Java se comunican con las bases de datos y manipulan sus datos utilizando la API JDBC. Un controlador de JDBC permite a las aplicaciones de Java conectarse a una base de datos en un DBMS específico, y nos permite obtener y manipular los datos de una base de datos.

Sección 25.2 Bases de datos relacionales

- Una base de datos relacional almacena los datos en tablas. Las tablas están compuestas de filas, y las filas están compuestas de columnas en las que se almacenan los valores.
- Una clave primaria proporciona un valor único que no puede duplicarse en otras filas.
- Cada columna de la tabla representa un atributo distinto.
- La clave primaria puede estar compuesta por más de una columna.
- SQL proporciona un conjunto completo de instrucciones que permiten a los programadores definir consultas complejas para recuperar datos de una base de datos.

- Toda columna en una clave primaria debe tener un valor, y el valor de la clave primaria debe ser único. A esto se le conoce como la Regla de integridad de entidades.
- Una relación de uno a varios entre tablas indica que una fila en una tabla puede tener muchas filas relacionadas en otra tabla.
- Una clave externa es una columna en una tabla que debe coincidir con el valor de la clave primaria en otra tabla.
- La clave externa ayuda a mantener la Regla de la integridad referencial: cada valor de clave externa debe aparecer como el valor de clave primaria de otra tabla. Las claves externas permiten que se una la información de varias tablas. Hay una relación de uno a varios entre una clave primaria y su correspondiente clave externa.

Sección 25.4.1 Consulta básica `SELECT`

- La forma básica de una consulta `SELECT` es:

  `SELECT * FROM` *nombreDeTabla*

  en donde el asterisco (*) indica que deben seleccionarse todas las columnas de *nombreDeTabla* y *nombreDeTabla* especifica la tabla en la base de datos de la cual se van a recuperar los datos.
- Para obtener ciertas columnas de una tabla, reemplace el asterisco (*) con una lista separada por comas de los nombres de las columnas.
- Los programadores procesan los resultados de una consulta conociendo de antemano el orden de las columnas en el resultado. Al especificar las columnas explícitamente, se garantiza que siempre se devuelvan en el orden especificado, incluso aunque el orden real en la(s) tabla(s) sea distinto.

Sección 25.4.2 La cláusula `WHERE`

- La cláusula `WHERE` opcional en una consulta `SELECT` especifica los criterios de selección para la consulta. La forma básica de una consulta `SELECT` con criterios de selección es:

  `SELECT` *nombreDeColumna1, nombreDeColumna2, …* `FROM` *nombreDeTabla* `WHERE` *criterios*

- La cláusula `WHERE` puede contener los operadores `<`, `>`, `<=`, `>=`, `=`, `<>` y `LIKE`. El operador `LIKE` se utiliza para relacionar patrones con los caracteres comodines por ciento (`%`) y guión bajo (`_`).
- Un carácter de por ciento (`%`) en un patrón indica que una cadena que coincida con ese patrón puede tener cero o más caracteres en la ubicación del carácter de por ciento en el patrón.
- Un guión bajo (`_`) en la cadena del patrón indica un carácter individual en esa posición en el patrón.

Sección 25.4.3 La cláusula `ORDER BY`

- El resultado de una consulta puede ordenarse en forma ascendente o descendente, mediante el uso de la cláusula `ORDER BY` opcional. La forma más simple de una cláusula `ORDER BY` es:

  `SELECT` *nombreDeColumna1, nombreDeColumna2, …* `FROM` *nombreDeTabla* `ORDER BY` *columna* `ASC`
  `SELECT` *nombreDeColumna1, nombreDeColumna2, …* `FROM` *nombreDeTabla* `ORDER BY` *columna* `DESC`

  en donde `ASC` especifica el orden ascendente, `DESC` especifica el orden descendente y *columna* especifica la columna en la cual se basa el ordenamiento. El orden predeterminado es ascendente, por lo que `ASC` es opcional.
- Pueden usarse varias columnas para ordenar mediante una cláusula `ORDER BY`, de la siguiente forma:

  `ORDER BY` *columna1 tipoDeOrden, columna2 tipoDeOrden*, …

- Las cláusulas `WHERE` y `ORDER BY` pueden combinarse en una consulta. Si se utiliza, la cláusula `ORDER BY` debe ser la última cláusula en la consulta.

Sección 25.4.4 Cómo fusionar datos de varias tablas: `INNER JOIN`

- Un operador `INNER JOIN` fusiona las filas de dos tablas al relacionar los valores en columnas que sean comunes para las dos tablas. La forma básica del operador `INNER JOIN` es:

```
SELECT nombreDeColumna1, nombreDeColumna2, …
FROM tabla1
INNER JOIN tabla2
   ON tabla1.nombreDeColumna = tabla2.nombreDeColumna
```

  La cláusula `ON` especifica las columnas de cada tabla que se van a comparar para determinar cuáles filas se van a unir. Si una instrucción de SQL utiliza columnas con el mismo nombre provenientes de varias tablas, los nombres de las

columnas deben estar completamente calificados y se debe colocar antes de ellos sus nombres de tablas y un operador punto (`.`).

Sección 25.4.5 La instrucción `INSERT`

- Una instrucción `INSERT` inserta una nueva fila en una tabla. La forma básica de esta instrucción es:

  `INSERT INTO` *nombreDeTabla* `(` *nombreDeColumna1, nombreDeColumna2, …, nombreDeColumnaN* `)`
  `VALUES` `(` *valor1, valor2, ..., valorN* `)`

  en donde *nombreDeTabla* es la tabla en la que se va a insertar la fila. El *nombreDeTabla* va seguido de una lista separada por comas de los nombres de columnas, entre paréntesis. La lista de nombres de columnas va seguida por la palabra clave `VALUES` de SQL, y una lista separada por comas de valores entre paréntesis.
- Las instrucciones de SQL utilizan el carácter de comilla sencilla (`'`) como delimitador para las cadenas. Para especificar una cadena que contenga una comilla sencilla en una instrucción de SQL, la comilla sencilla debe escaparse con otra comilla sencilla.

Sección 25.4.6 La instrucción UPDATE

- Una instrucción `UPDATE` modifica los datos en una tabla. La forma básica de la instrucción `UPDATE` es:

  `UPDATE` *nombreDeTabla*
  `SET` *nombreDeColumna1 = valor1, nombreDeColumna2 = valor2, …, nombreDeColumnaN = valorN*
  `WHERE` *criterios*

  en donde *nombreDeTabla* es la tabla de la que se van a actualizar los datos. El *nombreDeTabla* va seguido por la palabra clave `SET` y una lista separada por comas de los pares nombre/valor de las columnas, en el formato *nombreDeColumna=valor*. Los *criterios* de la cláusula `WHERE` determinan las filas que se van a actualizar.

Sección 25.4.7 La instrucción `DELETE`

- Una instrucción `DELETE` elimina filas de una tabla. La forma más simple de una instrucción `DELETE` es:

  `DELETE FROM` *nombreDeTabla* `WHERE` *criterios*

  en donde *nombreDeTabla* es la tabla de la que se va a eliminar una fila (o filas). Los *criterios* de la cláusula `WHERE` opcional determinan cuál(es) fila(s) se va(n) a eliminar. Si se omite esta cláusula, se eliminan todas las filas de la tabla.

Sección 25.8.1 Cómo conectarse y realizar consultas en una base de datos

- El paquete `java.sql` contiene clases e interfaces para manipular bases de datos relacionales en Java.
- Un objeto que implementa a la interfaz `Connection` administra la conexión entre el programa de Java y una base de datos. Los objetos `Connection` permiten a los programas crear instrucciones de SQL para acceder a los datos.
- El método `getConnection` de la clase `DriverManager` trata de conectarse a una base de datos especificada por su argumento URL. El URL ayuda al programa a localizar la base de datos. El URL incluye el protocolo y el subprotocolo de comunicación, junto con el nombre de la base de datos.
- El método `createStatement` de `Connection` crea un objeto de tipo `Statement`. El programa utiliza al objeto `Statement` para enviar instrucciones de SQL a la base de datos.
- El método `executeQuery` de `Statement` ejecuta una consulta y devuelve un objeto que implementa a la interfaz `ResultSet` que contiene el resultado de la consulta. Los métodos de `ResultSet` permiten a un programa manipular el resultado de la consulta.
- Un objeto `ResultSetMetaData` describe el contenido de un objeto `ResultSet`. Los programas pueden usar metadatos mediante la programación, para obtener información acerca de los nombres y tipos de las columnas del objeto `ResultSet`.
- El método `getColumnCount` de `ResultSetMetaData` recupera el número de columnas en el objeto `ResultSet`.
- El método `next` de `ResultSet` posiciona el cursor de `ResultSet` en la siguiente fila del objeto `ResultSet`. El cursor apunta a la fila actual. El método `next` devuelve el valor `boolean` `true` si el cursor puede posicionarse en la siguiente fila; en caso contrario el método devuelve `false`. Este método debe llamarse para empezar a procesar un objeto `ResultSet`.
- Al procesar objetos `ResultSet`, es posible extraer cada columna del objeto `ResultSet` como un tipo de Java específico. El método `getColumnType` de `ResultSetMetaData` devuelve un valor entero constante de la clase `Types` (paquete `java.sql`), indicando el tipo de los datos de una columna específica.

- Los métodos *obtener (get)* de `ResultSet` generalmente reciben como argumento un número de columna (como un valor `int`) o un nombre de columna (como un valor `String`), indicando cuál valor de la columna se va a obtener.
- Los números de fila y columna de un objeto `ResultSet` empiezan en 1.
- Cada objeto `Statement` puede abrir solamente un objeto `ResultSet` en un momento dado. Cuando un objeto `Statement` devuelve un nuevo objeto `ResultSet`, el objeto `Statement` cierra el objeto `ResultSet` anterior.
- El método `createStatement` de `Connection` tiene una versión sobrecargada que toma dos argumentos: el tipo y la concurrencia del resultado. El tipo del resultado especifica si el cursor del objeto `ResultSet` puede desplazarse en ambas direcciones o solamente hacia delante, y si el objeto `ResultSet` es susceptible a los cambios. La concurrencia del resultado especifica si el objeto `ResultSet` puede actualizarse con los métodos de actualización de `ResultSet`.
- Algunos controladores JDBC no soportan objetos `ResultSet` desplazables y/o actualizables.

Sección 25.8.2 Consultas en la base de datos `libros`

- El método `getColumnClass` de `TableModel` devuelve un objeto `Class` que representa a la superclase de todos los objetos en una columna específica. El objeto `JTable` utiliza esta información para configurar el desplegador de celdas y el editor de celdas predeterminados para esa columna en un objeto `JTable`.
- El método `getColumnClassName` de `ResultSetMetaData` obtiene el nombre completamente calificado de la columna especificada.
- El método `getColumnCount` de `TableModel` devuelve el número de columnas en el objeto `ResultSet` subyacente del modelo.
- El método `getColumnName` de `TableModel` devuelve el nombre de la columna en el objeto `ResultSet` subyacente del modelo.
- El método `getColumnName` de `ResultSetMetaData` obtiene el nombre de la columna proveniente del objeto `ResultSet`.
- El método `getRowCount` de `TableModel` devuelve el número de filas en el objeto `ResultSet` subyacente del modelo.
- El método `getValueAt` de `TableModel` devuelve el objeto `Object` en una fila y columna específicas del objeto `Result-Set` subyacente del modelo.
- El método `absolute` de `ResultSet` posiciona el cursor de `ResultSet` en una fila específica.
- El método `fireTableStructureChanged` de `AbstractTableModel` notifica a cualquier objeto `JTable` que utilice un objeto `TableModel` específico como su modelo, que los datos en el modelo han cambiado.

Sección 25.9 La interfaz `RowSet`

- La interfaz `RowSet` configura la conexión a la base de datos y ejecuta la consulta en forma automática.
- Hay dos tipos de objetos `RowSet`: conectados y desconectados.
- Un objeto `RowSet` conectado se conecta a la base de datos una vez, y permanece conectado hasta que termina la aplicación.
- Un objeto `RowSet` desconectado se conecta a la base de datos, ejecuta una consulta para obtener los datos de la base de datos y después cierra la conexión.
- `JdbcRowSet`, un objeto `RowSet` conectado, actúa como envoltura para un objeto `ResultSet` y nos permite desplazar y actualizar las filas en el objeto `ResultSet`. A diferencia de un objeto `ResultSet`, un objeto `JdbcRowSet` puede desplazarse y actualizarse de manera predeterminada.
- `CachedRowSet`, un objeto `RowSet` desconectado, coloca en caché de memoria los datos de un objeto `ResultSet`. Al igual que `JdbcRowSet`, un objeto `CachedRowSet` puede desplazarse y actualizarse. Un objeto `CachedRowSet` también es serializable, por lo que se puede pasar entre aplicaciones de Java a través de una red, como Internet.

Sección 25.10 Java DB/Apache Derby

- A partir del JDK 6.0, Sun Microsystems incluye la base de datos de código fuente abierto, basada exclusivamente en Java, llamada Java DB (la versión producida por Sun de Apache Derby) junto con el JDK.
- Java DB tiene una versión incrustada y una versión en red.

Sección 25.11 Objetos `PreparedStatement`

- La interfaz `PreparedStatement` nos permite crear instrucciones de SQL compiladas, que se ejecutan con más eficiencia que los objetos `Statement`.
- Las instrucciones `PreparedStatement` también pueden especificar parámetros, lo cual las hace más flexibles que las instrucciones `Statement`. Los programas pueden ejecutar la misma consulta varias veces, con distintos valores para los parámetros.

- Un parámetro se especifica mediante un signo de interrogación (?) en la instrucción SQL. Antes de ejecutar una instrucción PreparedStatement, el programa debe especificar los valores de los parámetros mediante el uso de los métodos *establecer (set)* de la interfaz PreparedStatement.
- El primer argumento del método setString de PreparedStatement representa el número del parámetro que se va a establecer, y el segundo argumento es el valor de ese parámetro.
- Los números de los parámetros se cuentan a partir de 1, empezando con el primer signo de interrogación (?).
- El método setString escapa de manera automática los valores de los parámetros String, según sea necesario.
- La interfaz PreparedStatement proporciona métodos establecer para cada tipo de SQL soportado.
- Una columna de identidad es la manera estándar en SQL de representar una columna autoincrementada. La palabra clave IDENTITY de SQL se utiliza para marcar una columna como columna de identidad.

Sección 25.12 Procedimientos almacenados

- JDBC permite a los programas invocar procedimientos almacenados mediante el uso de objetos que implementen a la interfaz CallableStatement.
- Los objetos CallableStatement pueden especificar parámetros de entrada, como PreparedStatement. Además, los objetos CallableStatement pueden especificar parámetros de salida, en los cuales un procedimiento almacenado puede colocar valores de retorno.

Sección 25.13 Procesamiento de transacciones

- El procesamiento de transacciones permite a un programa que interactúa con una base de datos, tratar una operación en la base de datos (o un conjunto de operaciones) como una sola operación. Dicha operación se conoce también como una operación atómica de una transacción.
- Al final de una transacción, se puede tomar una de dos decisiones: confirmar la transacción o rechazarla.
- Al confirmar la transacción se finaliza(n) toda(s) la(s) operación(es) en la base de datos; todas las inserciones, actualizaciones y eliminaciones que se llevan a cabo como parte de la transacción no pueden invertirse sin realizar una operación nueva en la base de datos.
- Al rechazar una transacción, la base de datos queda en el estado anterior al de la operación. Esto es útil cuando una parte de una transacción no se completa en forma apropiada.
- Java proporciona el procesamiento de transacciones a través de los métodos de la interfaz Connection.
- El método setAutoCommit especifica si cada instrucción de SQL se confirma al momento de completarse (un argumento true), o si deben agruparse varias instrucciones de SQL como una transacción (un argumento false).
- Si el argumento para setAutoCommit es false, el programa debe colocar, después la última instrucción SQL en la transacción, una llamada al método commit de Connection (para confirmar los cambios a la base de datos), o al método rollback de Connection (para regresar la base de datos al estado anterior a la transacción).
- El método getAutoCommit determina el estado de autoconfirmación para el objeto Connection.

Terminología

%, carácter comodín de SQL
_, carácter comodín de SQL
*, carácter comodín de SQL
absolute, método de ResultSet
AbstractTableModel, clase
addTableModelListener, método de la interfaz TableModel
base de datos
base de datos relacional
CachedRowSet, interfaz
CallableStatement, interfaz
clave externa
clave primaria
close, método de Connection
close, método de Statement
coincidencia de parámetros
columna
com.mysql.jdbc.Driver
conectarse a una base de datos
Connection, interfaz
consultar una base de datos
controlador de JDBC
createStatement, método de la interfaz Connection
criterios de selección
DELETE, instrucción de SQL
deleteRow, método de ResultSet
descubrimiento automático de controladores
DriverManager, clase
execute, método de la interfaz JdbcRowSet
execute, método de la interfaz Statement
executeQuery, método de la interfaz Statement
executeUpdate, método de la interfaz Statement
Fila
filtrar los datos en un objeto TableModel
fireTableStructureChanged, método de la clase AbstractTableModel

getColumnClass, método de la interfaz TableModel
getColumnClassName, método de la interfaz ResultSetMetaData
getColumnCount, método de la interfaz ResultSetMetaData
getColumnCount, método de la interfaz TableModel
getColumnName, método de la interfaz ResultSetMetaData
getColumnName, método de la interfaz TableModel
getColumnType, método de la interfaz ResultSetMetaData
getConnection, método de la clase DriverManager
getMetaData, método de la interfaz ResultSet
getMoreResults, método de la interfaz Statement
getObject, método de la interfaz ResultSet
getResultSet, método de la interfaz Statement
getRow, método de la interfaz ResultSet
getRowCount, método de la interfaz TableModel
getUpdateCount, método de la interfaz Statement
getValueAt, método de la interfaz TableModel
INNER JOIN, operador de SQL
INSERT, instrucción de SQL
insertRow, método de la interfaz ResultSet
java.sql, paquete
javax.sql, paquete
javax.sql.rowset, paquete
javax.swing.table, paquete
JDBC
JdbcRowSet, interfaz
JdbcRowSetImpl, clase
last, método de la interfaz ResultSet
metadatos
moveToCurrentRow, método de ResultSet
moveToInsertRow, método de ResultSet
MySQL Connector/J
MySQL, base de datos
next, método de la interfaz ResultSet
objeto ResultSet actualizable
objeto RowSet conectado
objeto RowSet desconectado
ON, cláusula
ordenamiento de filas
ORDER BY, cláusula
parámetro de salida
PreparedStatement, interfaz
procedimiento almacenado
regexFilter, método de la clase RowFilter
Regla de integridad de entidades
Regla de integridad referencial
relación de uno a varios
removeTableModelListener, método de la interfaz TableModel
resultado
ResultSet, interfaz
ResultSetMetaData, interfaz
RowFilter, clase
secuencia de comandos de SQL
SELECT, palabra clave de SQL
setCommand, método de la interfaz JdbcRowSet
setPassword, método de la interfaz JdbcRowSet
setRowFilter, método de la clase JTable
setRowSorter, método de la clase JTable
setString, método de la interfaz PreparedStatement
setUrl, método de la interfaz JdbcRowSet
setUsername, método de la interfaz JdbcRowSet
SQL (Lenguaje de consulta estructurado)
SQLException, clase
Statement, interfaz
sun.jdbc.odbc.JdbcOdbcDriver
tabla
TableModel, interfaz
TableModelEvent, clase
TableRowSorter, clase
Types, clase
unir
UPDATE, instrucción de SQL
updateRow, método de la interfaz ResultSet
WHERE, cláusula de una instrucción de SQL

Ejercicios de autoevaluación

25.1 Complete las siguientes oraciones:

a) El lenguaje de consulta de bases de datos estándar internacional es ________.
b) Una tabla en una base de datos consiste de ________ y ________.
c) Los objetos Statement devuelven los resultados de una consulta de SQL como objetos ________.
d) La ________ identifica en forma única a cada fila en una tabla.
e) La palabra clave ________ de SQL va seguida por los criterios de selección que especifican las filas a seleccionar en una consulta.
f) Las palabras clave ________ de SQL especifican la manera en que se ordenan las filas en una consulta.
g) Al proceso de fusionar filas de varias tablas de una base de datos se le conoce como ________ las tablas.
h) Una ________ es una colección organizada de datos.
i) Una ________ es un conjunto de columnas cuyos valores coinciden con los valores de clave primaria de otra tabla.

j) Un objeto ______________ se utiliza para obtener una conexión (`Connection`) a una base de datos.
k) La interfaz ______________ ayuda a administrar la conexión entre un programa de Java y una base de datos.
l) Un objeto ______________ se utiliza para enviar una consulta a una base de datos.
m) A diferencia de un objeto `ResultSet`, los objetos ______________, ______________ y ______________ son desplazables y se pueden actualizar de manera predeterminada.
n) ______________, un objeto `RowSet` desconectado, coloca en caché los datos de un objeto `ResultSet` en la memoria.

Respuestas a los ejercicios de autoevaluación

25.1 a) SQL. b) filas, columnas. c) `ResultSet`. d) clave primaria. e) `WHERE`. f) `ORDER BY`. g) unir. h) base de datos. i) clave externa. j) `DriverManager`. k) `Connection`. l) `Statement`. m) `JdbcRowSet`, `CachedRowSet`. n) `CachedRowSet`.

Ejercicios

25.2 Utilizando las técnicas mostradas en este capítulo, defina una aplicación de consulta completa para la base de datos `libros`. Proporcione las siguientes consultas predefinidas:

a) Seleccionar todos los autores de la tabla `autores`.
b) Seleccionar un autor específico y mostrar todos los libros de ese autor. Incluir el título, año e ISBN. Ordenar la información alfabéticamente, en base al apellido paterno y nombre de pila del autor.
c) Seleccionar una editorial específica y mostrar todos los libros publicados por esa editorial. Incluir el título, año e ISBN. Ordenar la información alfabéticamente por título.
d) Proporcione cualquier otra consulta que usted crea que sea apropiada.
Muestre un objeto `JComboBox` con nombres apropiados para cada consulta predefinida. Además, permita a los usuarios suministrar sus propias consultas.

25.3 Defina una aplicación de manipulación de bases de datos para la base de datos `libros`. El usuario deberá poder editar los datos existentes y agregar nuevos datos a la base de datos (obedeciendo las restricciones de integridad referencial y de entidades). Permita al usuario editar la base de datos de las siguientes formas:

a) Agregar un nuevo autor.
b) Editar la información existente para un autor.
d) Agregar un nuevo título para un autor. (Recuerde que el libro debe tener una entrada en la tabla `isbnAutor`).
d) Agregar una nueva entrada en la tabla `isbnAutor` para enlazar los autores con los títulos.

25.4 En la sección 10.7 presentamos una jerarquía nómina-empleado para calcular la nómina de cada empleado. En este ejercicio proporcionamos una base de datos de empleados que corresponde a la jerarquía nómina-empleado. (En el CD que acompaña a este libro y en nuestro sitio Web `www.deitel.com`, se proporciona una secuencia de comandos de SQL para crear la base de datos de empleados, junto con los ejemplos de este capítulo). Escriba una aplicación que permita al usuario:

a) Agregar empleados a la tabla `empleado`.
b) Para cada nuevo empleado, agregar información de la nómina a la tabla correspondiente. Por ejemplo, para un empleado asalariado agregue la información de nómina a la tabla `empleadosAsalariados`.

La figura 25.33 es el diagrama de relaciones de entidades para la base de datos `empleados`.

25.5 Modifique el ejercicio 25.4 para proporcionar un objeto `JComboBox` y un objeto `JTextArea` para permitir al usuario realizar una consulta que se seleccione del objeto `JComboBox`, o que se defina en el objeto `JTextArea`. Las consultas predefinidas de ejemplo son:

a) Seleccionar a todos los empleados que trabajen en el departamento de VENTAS.
b) Seleccionar a los empleados por horas que trabajen más de 30 horas.
c) Seleccionar a todos los empleados por comisión en orden descendente en base a la tasa de comisión.

25.6 Modifique el ejercicio 25.5 para realizar las siguientes tareas:

a) Incrementar el salario base en un 10% para todos los empleados base más comisión.
b) Si el cumpleaños del empleado es en el mes actual, agregar un bono de $100.
c) Para todos los empleados por comisión cuyas ventas brutas sean mayores de $10,000, agregar un bono de $100.

25.7 Modifique el programa de las figuras 25.31 a 25.33 para proporcionar un objeto `JButton` que permita al usuario llamar a un método `actualizarPersona` en la interfaz `ConsultasPersona`, para actualizar la entrada actual en la base de datos `LibretaDirecciones`.

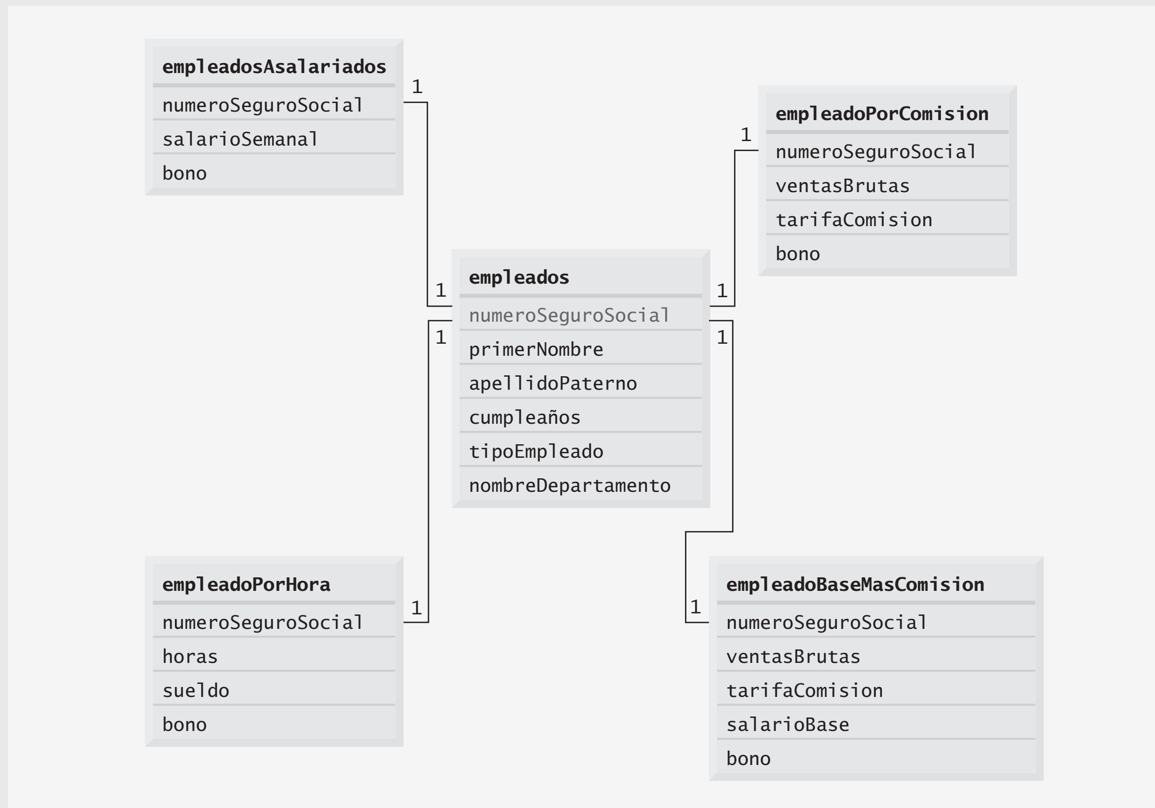
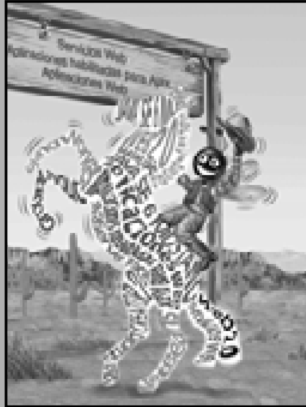


Figura 25.33 | Relaciones de las tablas en `empleados`.

26

Aplicaciones Web: parte 1



Si cualquier hombre prepara su caso y coloca su nombre al pie de la primera página, yo le daré una respuesta inmediata. Si me obliga a dar vuelta a la hoja, deberá esperar a mi conveniencia.

—Lord Sandwich

Regla uno: nuestro cliente siempre tiene la razón. Regla dos: si piensas que nuestro cliente está mal, consulta la Regla uno.

—Anónimo

Una pregunta justa debe ir seguida de un acto en silencio.

—Dante Alighieri

Vendrás aquí y obtendrás libros que abrirán tus ojos, oídos y tu curiosidad; y sacarán tu interior, o meterán tu exterior.

—Ralph Waldo Emerson

OBJETIVOS

En este capítulo aprenderá a:

- Desarrollar aplicaciones Web mediante el uso de las tecnologías de Java y Java Studio Creator 2.0.
- Crear JavaServer Pages con componentes JavaServer Faces.
- Crear aplicaciones Web que consistan de varias páginas.
- Validar la entrada del usuario en una página Web.
- Mantener la información de estado acerca de un usuario, con rastreo de sesión y cookies.

Plan general

26.1 Introducción

En este capítulo, presentaremos el **desarrollo de aplicaciones Web** con tecnología basada en Java. Las aplicaciones basadas en Web crean contenido Web para los clientes navegadores Web. Este contenido Web incluye el Lenguaje de marcado de hipertexto extensible (XHTML), las secuencias de comandos del lado servidor, imágenes y datos binarios. Para aquellos que no están familiarizados con XHTML, en el CD que se incluye con este libro hay tres capítulos de nuestro libro *Internet & World Wide Web How to Program, 3/e* [Introduction to XHTML: Part 1, Introduction to XHTML: Part 2 y Cascading Style Sheets (CSS)]. En los capítulos 26 a 28, vamos a suponer que usted ya sabe utilizar XHTML.

Este capítulo empieza con las generalidades de la arquitectura de aplicaciones multinivel, y las tecnologías Web de Java para implementar aplicaciones multinivel. Después presentaremos varios ejemplos que demuestran el desarrollo de aplicaciones Web. El primer ejemplo lo introducirá al desarrollo Web de Java. En el segundo ejemplo, crearemos una aplicación Web que simplemente muestra la apariencia visual de varios componentes de GUI de aplicaciones Web. Después, le demostraremos cómo utilizar los componentes de validación y los métodos de validación personalizados para asegurar que la entrada del usuario sea válida antes de enviarla para que el servidor la procese. El capítulo termina con dos ejemplos acerca de cómo personalizar la experiencia de un usuario mediante el rastreo de sesiones.

En el capítulo 27 continuaremos nuestra discusión acerca del desarrollo de aplicaciones Web con conceptos más avanzados, incluyendo los componentes habilitados para AJAX del modelo de programación Java BluePrints de Sun. AJAX ayuda a las aplicaciones basadas en Web a proporcionar la interactividad y capacidad de respuesta que los usuarios esperan comúnmente de las aplicaciones de escritorio.

A lo largo de este capítulo y del capítulo 27 utilizaremos **Sun Java Studio Creator 2.0**: un IDE que ayuda al programador a crear aplicaciones Web mediante el uso de tecnologías de Java, como JavaServer Pages y JavaServer Faces. Para implementar los ejemplos que se presentan en este capítulo, debe instalar Java Studio Creator 2.0, el

cual está disponible para su descarga en `developers.sun.com/prodtech/javatools/jscreator/downloads/index.jsp`. Las características de Java Studio Creator 2.0 se están incorporando en Netbeans 5.5, mediante un complemento llamado Netbeans Visual Web Pack 5.5 (`www.netbeans.org/products/visualweb/`).

26.2 Transacciones HTTP simples

El desarrollo de aplicaciones Web requiere una comprensión básica de las redes y de World Wide Web. En esta sección, hablaremos sobre el Protocolo de transferencia de hipertexto (HTTP) y lo que ocurre "tras bambalinas", cuando un usuario solicita una página Web en un navegador. HTTP especifica un conjunto de métodos y encabezados que permiten a los clientes y servidores interactuar e intercambiar información de una manera uniforme y confiable.

En su forma más simple, una página Web no es más que un documento XHTML: un archivo de texto simple que contiene **marcado** (es decir, **etiquetas**) para describir a un navegador Web cómo mostrar y dar formato a la información del documento. Por ejemplo, el siguiente marcado de XHTML:

```
<title>Mi pagina Web</title>
```

indica que el navegador debe mostrar el texto entre la **etiqueta inicial** `<title>` y la **etiqueta final** `</title>` en la barra de título del navegador. Los documentos XHTML también pueden contener datos de **hipertexto** (lo que se conoce comúnmente como **hipervínculos**) que vinculan a distintas páginas, o a otras partes de la misma página. Cuando el usuario activa un hipervínculo (por lo general, haciendo clic sobre él con el ratón), la página Web solicitada se carga en el navegador Web del usuario.

HTTP utiliza **URIs (Identificadores uniformes de recursos)** para identificar datos en Internet. Los URIs que especifican las ubicaciones de los documentos se llaman **URLs (Localizadores uniformes de recursos)**. Los URLs comunes se refieren a archivos, directorios u objetos que realizan tareas complejas, como búsquedas en bases de datos y en Internet. Si usted conoce el URL de HTTP de un documento XHTML disponible públicamente en cualquier parte en la Web, puede acceder a este documento a través de HTTP.

Un URL contiene la información que dirige a un navegador al recurso que el usuario desea utilizar. Las computadoras que ejecutan software de **servidor Web** hacen disponibles esos recursos. Vamos a examinar los componentes del URL:

```
http://www.deitel.com/libros/descargas.html
```

El `http://` indica que el recurso se debe obtener mediante el protocolo HTTP. La porción intermedia, `www.deitel.com`, es el **nombre del host** completamente calificado del servidor: el nombre del servidor en el que reside el recurso. Esta computadora se conoce comúnmente como **host**, debido a que aloja y da mantenimiento a los recursos. El nombre de host `www.deitel.com` se traduce en una **dirección IP** (68.236.123.125), la cual identifica al servidor así como un número telefónico identifica de forma única a una línea telefónica específica. Esta traducción se lleva a cabo mediante un **servidor del sistema de nombres de dominio (DNS)**: una computadora que mantiene una base de datos de nombres de host y sus correspondientes direcciones IP; a este proceso se le conoce como **búsqueda DNS (DNS lookup)**.

El resto del URL (es decir, `/libros/descargas.html`) especifica tanto el nombre del recurso solicitado (el documento XHTML llamado `descargas.html`) como su ruta o ubicación (`/libros`), en el servidor Web. La ruta podría especificar la ubicación de un directorio actual en el sistema de archivos del servidor Web. Sin embargo, por cuestiones de seguridad, la ruta generalmente especifica la ubicación de un **directorio virtual**. El servidor traduce el directorio virtual en una ubicación real en el servidor (o en otra computadora en la red del servidor), con lo cual se oculta la verdadera ubicación del recurso. Algunos recursos se crean en forma dinámica, por lo que no residen en ninguna parte del servidor. El nombre de host en el URL para dicho recurso especifica el servidor correcto; la ruta y la información sobre el recurso identifican la ubicación del recurso con el que se va a responder a la petición del cliente.

Cuando el navegador Web recibe un URL, realiza una transacción HTTP simple para obtener y mostrar la página Web que se encuentra en esa dirección. En la figura 26.1 se ilustra la transacción en forma detallada, mostrando la interacción entre el navegador Web (el lado cliente) y la aplicación servidor Web (el lado servidor).

En la figura 26.1, el navegador Web envía una petición HTTP al servidor. La petición (en su forma más simple) es

```
GET /libros/descargas.html HTTP/1.1
```

La palabra `GET` es un **método HTTP**, el cual indica que el cliente desea obtener un recurso del servidor. El resto de la petición proporciona el nombre de la ruta del recurso (un documento XHTML), junto con el nombre del protocolo y el número de versión (`HTTP/1.1`).

Cualquier servidor que entienda HTTP (versión 1.1) puede traducir esta petición y responder en forma apropiada. En la figura 26.2 se muestran los resultados de una petición exitosa. Primero, el servidor responde enviando una línea de texto que indica la versión de HTTP, seguida de un código numérico y una frase que describe el estado de la transacción. Por ejemplo,

```
HTTP/1.1 200 OK
```

indica que se tuvo éxito, mientras que

```
HTTP/1.1 404 Not found
```

informa al cliente que el servidor Web no pudo localizar el recurso solicitado. En la página `www.w3.org/Protocols/HTTP/HTRESP.html` encontrará una lista completa de códigos numéricos que indican el estado de una transacción HTTP.

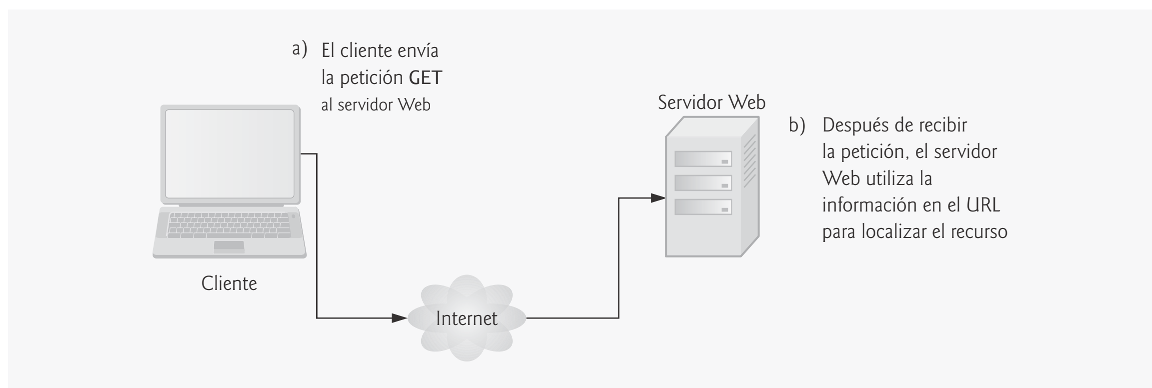


Figura 26.1 | Interacción entre el cliente y el servidor Web. *Paso 1:* la petición `GET`.

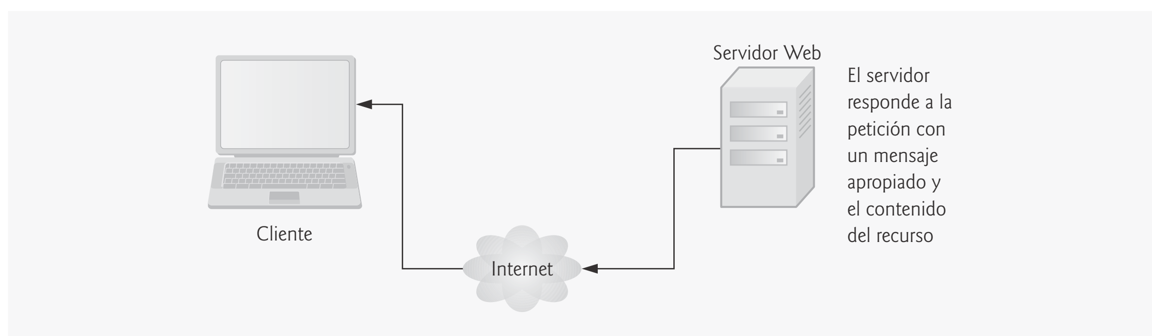


Figura 26.2 | Interacción entre el cliente y el servidor Web. *Paso 2:* la respuesta HTTP.

Después, el servidor envía uno o más **encabezados HTTP**, los cuales proporcionan información adicional sobre los datos que se van a enviar. En este caso, el servidor está enviando un documento de texto XHTML, por lo que el encabezado HTTP para este ejemplo sería:

```
Content-type: text/html
```

La información que se proporciona en este encabezado especifica el tipo de **Extensiones de correo Internet multipropósito (MIME)** del contenido que el servidor va a transmitir al navegador. MIME es un estándar de

Internet que especifica formatos de datos, para que los programas puedan interpretar los datos en forma correcta. Por ejemplo, el tipo MIME `text/plain` indica que la información enviada es texto que puede mostrarse directamente, sin interpretar el contenido como marcado de XHTML. De manera similar, el tipo MIME `image/jpeg` indica que el contenido es una imagen JPEG. Cuando el navegador recibe este tipo MIME, trata de mostrar la imagen.

El encabezado, o conjunto de encabezados, va seguido por una línea en blanco, la cual indica al cliente que el servidor terminó de enviar encabezados HTTP. Después, el servidor envía el contenido del documento XHTML solicitado (`descargas.html`). El servidor termina la conexión cuando se completa la transferencia del recurso. El navegador del lado cliente analiza el marcado de XHTML que recibe y **despliega** (o visualiza) los resultados.

26.3 Arquitectura de aplicaciones multinivel

Las aplicaciones basadas en Web son **aplicaciones multinivel** (comúnmente conocidas como **aplicaciones de *n* niveles**), que dividen la funcionalidad en **niveles** separados (es decir, agrupaciones lógicas de funcionalidad). Aunque los niveles pueden localizarse en la misma computadora, por lo general, los niveles de las aplicaciones basadas en Web residen en computadoras separadas. En la figura 26.3 se presenta la estructura básica de una **aplicación basada en Web de tres niveles**.

El **nivel inferior** (también conocido como Nivel de datos o nivel de información) mantiene los datos de la aplicación. Por lo general, este nivel almacena los datos en un sistema de administración de bases de datos relacionales (RDBMS). En el capítulo 25 hablamos sobre los sistemas RDBMS. Por ejemplo, una tienda podría tener una base de datos de información sobre el inventario, que contenga descripciones de productos, precios y cantidades en almacén. La misma base de datos podría también contener información sobre los clientes, como los nombres de usuarios, direcciones de facturación y números de tarjetas de crédito. Podría haber varias bases de datos residiendo en una o más computadoras, que en conjunto forman los datos de la aplicación.

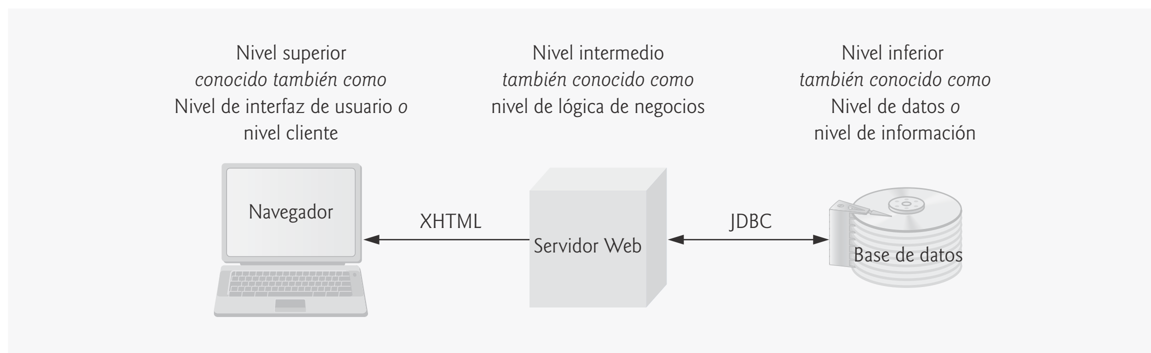


Figura 26.3 | Arquitectura de tres niveles.

El **nivel intermedio** implementa la lógica de negocios, de controlador y de presentación para controlar las interacciones entre los clientes de la aplicación y sus datos. El nivel intermedio actúa como intermediario entre los datos en el nivel de información y los clientes de la aplicación. La **lógica de control** del nivel intermedio procesa las peticiones de los clientes (como las peticiones para ver un catálogo de productos) y obtiene datos de la base de datos. Después, la **lógica de presentación** del nivel intermedio procesa los datos del nivel de información y presenta el contenido al cliente. Por lo general, las aplicaciones Web presentan datos a los clientes en forma de documentos XHTML.

La **lógica comercial** en el nivel intermedio hace valer las **reglas comerciales** y asegura que los datos sean confiables antes de que la aplicación servidor actualice la base de datos, o presente los datos a los usuarios. Las reglas comerciales dictan la forma en que los clientes pueden o no acceder a los datos de la aplicación, y la forma en que ésta procesa los datos. Por ejemplo, una regla comercial en el nivel intermedio de una aplicación basada en Web para una tienda podría asegurar que todas las cantidades de los productos sean siempre positivas. La petición de un cliente de establecer una cantidad negativa en la base de datos de información de productos del nivel inferior sería rechazada por la lógica comercial del nivel intermedio.

El **nivel superior** (nivel cliente) es la interfaz de usuario de la aplicación, la cual recopila los datos de entrada y de salida. Los usuarios interactúan en forma directa con la aplicación a través de la interfaz de usuario, que por lo general es el navegador Web, el teclado y el ratón. En respuesta a las acciones del usuario (por ejemplo, hacer clic en un hipervínculo), el nivel cliente interactúa con el nivel intermedio para hacer peticiones y obtener datos del nivel de información. Después, el nivel cliente muestra los datos obtenidos para el usuario. El nivel cliente nunca interactúa directamente con el nivel de información.

Las aplicaciones multinivel de Java se implementan comúnmente mediante el uso de las características de Java Enterprise Edition (Java EE). Las tecnologías que usaremos para desarrollar aplicaciones Web en los capítulos 26 a 28 son parte de Java EE 5 (`java.sun.com/javaee`).

26.4 Tecnologías Web de Java

Las tecnologías Web de Java evolucionan en forma continua, para ofrecer a los desarrolladores niveles mayores de abstracción, y una mayor separación de los niveles de la aplicación. Esta separación facilita el mantenimiento y la extensibilidad de las aplicaciones Web. Un diseñador gráfico puede crear la interfaz de usuario de la aplicación sin tener que preocuparse por la lógica de páginas subyacente, la cual estará a cargo de un programador. Mientras tanto, el programador está libre para enfocarse en la lógica comercial de la aplicación, dejando al diseñador los detalles sobre la construcción de una aplicación atractiva y fácil de usar. Java Studio Creator 2 es el paso más reciente en esta evolución, ya que nos permite desarrollar la GUI de una aplicación Web mediante una herramienta de diseño tipo "arrastrar y soltar", mientras que podemos manejar la lógica comercial en clases de Java separadas.

26.4.1 Servlets

Los servlets son la vista de nivel más bajo de las tecnologías de desarrollo en Java que veremos en este capítulo. Utilizan el modelo petición-respuesta HTTP de comunicación entre cliente y servidor.

Los **servlets** extienden la funcionalidad de un servidor, al permitir que éste genere contenido dinámico. Por ejemplo, los servlets pueden generar en forma dinámica documentos XHTML personalizados, ayudar a proporcionar un acceso seguro a un sitio Web, interactuar con bases de datos a beneficio de un cliente y mantener la información de sesión única para cada cliente. Un componente del servidor Web, conocido como **contenedor de servlets**, ejecuta los servlets e interactúa con ellos. Los paquetes **`javax.servlet`** y **`javax.servlet.http`** proporcionan las clases e interfaces para definir servlets. El contenedor de servlets recibe peticiones HTTP de un cliente y dirige cada petición al servlet apropiado. El servlet procesa la petición y devuelve una respuesta apropiada al cliente; por lo general en forma de un documento XHTML o **XML (Lenguaje de marcado extensible)** para mostrarlo en el navegador. XML es un lenguaje que se utiliza para intercambiar datos estructurados en la Web.

Desde el punto de vista arquitectónico, todos los servlets deben implementar a la interfaz **`Servlet`** del paquete `javax.servlet`, la cual asegura que cada servlet se pueda ejecutar en el marco de trabajo proporcionado por el contenedor de servlets. La interfaz `Servlet` declara métodos que el contenedor de servlets utiliza para administrar el ciclo de vida del servlet. Este ciclo de vida empieza cuando el contenedor de servlets lo carga en memoria; por lo general, en respuesta a la primera petición del servlet. Antes de que el servlet pueda manejar esa petición, el contenedor invoca al método `init` del servlet, el cual se llama sólo una vez durante el ciclo de vida de un servlet para inicializarlo. Una vez que `init` termina su ejecución, el servlet está listo para responder a su primera petición. Todas las peticiones se manejan mediante el **método `service`** de un servlet, el cual es el método clave para definir la funcionalidad de un servlet. El método `service` recibe la petición, la procesa y envía una respuesta al cliente. Durante el ciclo de vida de un servlet, se hace una llamada al método `service` por cada petición. Cada nueva petición se maneja comúnmente en un subproceso de ejecución separado (administrado por el contenedor de servlets), por lo que cada servlet debe ser seguro para los subprocesos. Cuando el contenedor de servlets termina el servlet (por ejemplo, cuando el contenedor de servlets necesita más memoria o cuando se cierra), se hace una llamada al método `destroy` del servlet para liberar los recursos que éste ocupa.

26.4.2 JavaServer Pages

La tecnología **JavaServer Pages (JSP)** es una extensión de la tecnología de los servlets. El contenedor de JSPs traduce cada JSP y la convierte en un servlet. A diferencia de los servlets, las JSPs nos ayudan a separar la presentación del contenido. Las JavaServer Pages permiten a los programadores de aplicaciones Web crear contenido dinámico mediante la reutilización de componentes predefinidos, y mediante la interacción con componentes que utilizan secuencias de comandos del lado servidor. Los programadores de JSPs pueden utilizar componen-

tes especiales de software llamados JavaBeans, y bibliotecas de etiquetas personalizadas que encapsulan una funcionalidad dinámica y compleja. Un **JavaBean** es un componente reutilizable que sigue ciertas convenciones para el diseño de clases. Por ejemplo, las clases de JavaBeans que permiten operaciones de lectura y escritura de variables de instancias deben proporcionar métodos *obtener (get)* y *establecer (set)* apropiados. El conjunto completo de convenciones de diseño de clases se describe en la especificación de los JavaBeans (`java.sun.com/products/javabeans/glasgow/index.html`).

Bibliotecas de etiquetas personalizadas

Las **bibliotecas de etiquetas personalizadas** son una poderosa característica de la tecnología JSP, que permite a los desarrolladores de Java ocultar el código para acceder a una base de datos y otras operaciones complejas mediante **etiquetas personalizadas**. Para usar dichas herramientas, sólo tenemos que agregar las etiquetas personalizadas a la página. Esta simpleza permite a los diseñadores de páginas Web, que no estén familiarizados con Java, mejorar las páginas Web con poderoso contenido dinámico y capacidades de procesamiento. Las clases e interfaces de JSP se encuentran en los paquetes `javax.servlet.jsp` y `javax.servlet.jsp.tagext`.

Componentes de JSP

Hay cuatro componentes clave para las JSPs: directivas, acciones, elementos de secuencia de comandos y bibliotecas de etiquetas. Las **directivas** son mensajes para el **contenedor de JSPs**: el componente del servidor Web que ejecuta las JSPs. Las directivas nos permiten especificar configuraciones de páginas, para incluir contenido de otros recursos y especificar bibliotecas de etiquetas personalizadas para usarlas en las JSPs. Las **acciones** encapsulan la funcionalidad en etiquetas predefinidas que los programadores pueden incrustar en JSPs. A menudo, las acciones se realizan con base en la información que se envía al servidor como parte de una petición específica de un cliente. También pueden crear objetos de Java para usarlos en las JSPs. Los **elementos de secuencia de comandos** permiten al programador insertar código que interactúe con los componentes en una JSP (y posiblemente con otros componentes de aplicaciones Web) para realizar el procesamiento de peticiones. Las **bibliotecas de etiquetas** forman parte del **mecanismo de extensión de etiquetas** que permite a los programadores crear etiquetas personalizadas. Dichas etiquetas permiten a los diseñadores de páginas Web manipular el contenido de las JSPs sin necesidad de tener un conocimiento previo sobre Java. La **Biblioteca de etiquetas estándar de JavaServer Pages (JSTL)** proporciona la funcionalidad para muchas tareas de aplicaciones Web comunes, como iterar a través de una colección de objetos y ejecutar instrucciones de SQL.

Contenido estático

Las JSPs pueden contener otro tipo de contenido estático. Por ejemplo, las JSPs comúnmente incluyen marcado XHTML o XML. A dicho marcado se le conoce como **datos de plantilla fija** o **texto de plantilla fija**. Cualquier texto literal en una JSP se traduce en una literal `String` en la representación de la JSP en forma de servlet.

Procesamiento de una petición de JSP

Cuando un servidor habilitado para JSP recibe la primera petición para una JSP, el contenedor de JSPs traduce esa JSP en un servlet, el cual maneja la petición actual y las futuras peticiones a esa JSP. Por lo tanto, las JSPs se basan en el mismo mecanismo de petición-respuesta que los servlets para procesar las peticiones de los clientes, y enviar las respuestas.



Tip de rendimiento 26.1

Algunos contenedores de JSPs traducen las JSPs en servlets al momento de desplegar las JSPs (es decir, cuando la aplicación se coloca en un servidor Web). Esto elimina la sobrecarga de la traducción para el primer cliente que solicita cada JSP, ya que la JSP se traducirá antes de que un cliente la haya solicitado.

26.4.3 JavaServer Faces

JavaServer Faces (JSF) es un marco de trabajo para aplicaciones Web que simplifica el diseño de la interfaz de usuario de una aplicación, y separa aún más la presentación de una aplicación Web de su lógica comercial. Un **marco de trabajo (framework)** simplifica el desarrollo de aplicaciones, al proporcionar bibliotecas y (algunas veces) herramientas de software para ayudar al programador a organizar y crear sus aplicaciones. Aunque el marco

de trabajo JSF puede usar muchas tecnologías para definir las páginas en las aplicaciones Web, este capítulo se enfoca en las aplicaciones JSF que utilizan JavaServer Pages. JSF proporciona un conjunto de componentes de interfaz de usuario, o **componentes de JSF** que simplifican el diseño de páginas Web. Estos componentes son similares a los componentes de Swing que se utilizan para crear aplicaciones con GUI. JSF proporciona dos bibliotecas de etiquetas personalizadas de JSP para agregar estos componentes a una página JSP. JSF también incluye APIs para manejar eventos de componentes (como el procesamiento de los cambios de estado de los componentes y la validación de la entrada del usuario), navegar entre las páginas de una aplicación Web y mucho más. El programador diseña la apariencia visual de una página con JSF, agregando etiquetas a un archivo JSP y manipulando sus atributos. El programador define el comportamiento de la página por separado, en un archivo de código fuente de Java relacionado.

Aunque los componentes estándar de JSF son suficientes para la mayoría de las aplicaciones Web básicas, también podemos escribir bibliotecas de componentes personalizados. Hay bibliotecas de componentes adicionales, disponibles en el proyecto **Java BluePrints**, el cual muestra las mejores prácticas para desarrollar aplicaciones en Java. Muchos otros distribuidores ofrecen bibliotecas de componentes de JSF. Por ejemplo, Oracle proporciona alrededor de 100 componentes en su biblioteca ADF Faces. Aquí hablaremos sobre una de esas bibliotecas de componentes, conocida como BluePrints AJAX (`blueprints.dev.java.net/ajaxcomponents.html`). En el siguiente capítulo hablaremos sobre los componentes de Java BluePrints para crear aplicaciones de JSF habilitadas para AJAX.

26.4.4 Tecnologías Web en Java Studio Creator 2

Las aplicaciones Web de Java Studio Creator 2 consisten en una o más páginas Web JSP, integradas en el marco de trabajo JavaServer Faces. Estos archivos JSP tienen la extensión de archivo **`.jsp`** y contienen los elementos de la GUI de la página Web. Las JSPs también pueden contener JavaScript para agregar funcionalidad a la página. Las JSPs se pueden personalizar en Java Studio Creator 2 al agregar componentes de JSF, incluyendo etiquetas, campos de texto, imágenes, botones y otros componentes de GUI. El IDE nos permite diseñar las páginas en forma visual, al arrastrar y soltar estos componentes en una página; también podemos personalizar una página Web al editar el archivo `.jsp` en forma manual.

Cada archivo JSP que se crea en Java Studio Creator 2 representa una página Web, y tiene su correspondiente clase JavaBean, denominada **bean de página**. Una clase JavaBean debe tener un constructor predeterminado (o sin argumentos), junto con métodos *obtener (get)* y *establecer (set)* para todas las propiedades del bean (es decir, las variables de instancia). El bean de página define las propiedades para cada uno de los elementos de la página. El bean de página también contiene los manejadores de eventos y los métodos de ciclo de vida de las páginas para administrar tareas, como la inicialización y despliegue de las páginas, y demás código de soporte para la aplicación Web.

Toda aplicación Web creada con Java Studio Creator 2 tiene otros tres JavaBeans. El objeto **`RequestBean`** se mantiene en **ámbito de petición**; este objeto existe sólo mientras dure una petición HTTP. Un objeto **`SessionBean`** tiene **ámbito de sesión**; el objeto existe durante una sesión de navegación del usuario, o hasta que se agota el tiempo de la sesión. Hay un único objeto `SessionBean` para cada usuario. Por último, el objeto **`ApplicationBean`** tiene **ámbito de aplicación**; este objeto es compartido por todas las instancias de una aplicación y existe mientras que la aplicación esté desplegada en un servidor Web. Este objeto se utiliza para almacenar datos a nivel de aplicación o para procesamiento; sólo existe una instancia para la aplicación, sin importar el número de sesiones abiertas.

26.5 Creación y ejecución de una aplicación simple en Java Studio Creator 2

Nuestro primer ejemplo muestra la hora del día del servidor Web en una ventana del navegador. Al ejecutarse, este programa muestra el texto `"Hora actual en el servidor Web"`, seguido de la hora del servidor Web. La aplicación contiene una sola página Web y, como mencionamos antes, consiste de dos archivos relacionados: un archivo JSP (figura 26.4) y un archivo de bean de página de soporte (figura 26.6). La aplicación tiene también los tres beans de datos con ámbito para los ámbitos de petición, sesión y aplicación. Como esta aplicación no almacena datos, estos beans no se utilizan en este ejemplo. Primero hablaremos sobre el marcado en el archivo JSP, el código en el archivo de bean de página y la salida de la aplicación; después proporcionaremos instrucciones

detalladas para crear el programa. [*Nota:* el marcado en la figura 26.4 y en los demás listados de archivos JSP en este capítulo es el mismo que el marcado que aparece en Java Studio Creator 2, pero hemos cambiado el formato de estos listados para fines de presentación, para que el código sea más legible].

Java Studio Creator 2 genera todo el marcado que se muestra en la figura 26.4 cuando establecemos el título de la página Web, arrastramos dos componentes **Texto estático** en la página y establecemos las propiedades de estos componentes. Los componentes `Texto estático` muestran texto que el usuario no puede editar. En breve le mostraremos estos pasos.

```
1  <?xml version = "1.0" encoding = "UTF-8"?>
2
3  <!-- Fig. 26.4: Hora.jsp -->
4  <!-- Archivo JSP generado por Java Studio Creator 2, que muestra -->
5  <!-- la hora actual en el servidor Web -->
6  <jsp:root version = "1.2"
7     xmlns:f = "http://java.sun.com/jsf/core"
8     xmlns:h = "http://java.sun.com/jsf/html"
9     xmlns:jsp = "http://java.sun.com/JSP/Page"
10    xmlns:ui = "http://www.sun.com/web/ui">
11    <jsp:directive.page contentType = "text/html;charset = UTF-8"
12       pageEncoding = "UTF-8"/>
13    <f:view>
14       <ui:page binding = "#{Hora.page1}" id = "page1">
15          <ui:html binding = "#{Hora.html1}" id = "html1">
16             <ui:head binding = "#{Hora.head1}" id = "head1"
17                title = "Hora Web: un ejemplo simple">
18                <ui:link binding = "#{Hora.link1}" id = "link1"
19                   url = "/resources/stylesheet.css"/>
20                </ui:head>
21                <ui:meta content = "60" httpEquiv = "refresh" />
22                <ui:body binding = "#{Hora.body1}" id = "body1"
23                   style = "-rave-layout: grid">
24                   <ui:form binding = "#{Hora.form1}" id = "form1">
25                      <ui:staticText binding = "#{Hora.encabezadoHora}" id =
26                         "encabezadoHora" style = "font-size: 18px; left: 24px;
27                         top: 24px; position: absolute" text = "Hora actual
28                         en el servidor Web : "/>
29                      <ui:staticText binding = "#{Hora.textoReloj}" id =
30                         "textoReloj" style = "background-color: black;
31                         color: yellow; font-size: 18px; left: 24px; top:
32                         48px; position: absolute"/>
33                </ui:form>
34             </ui:body>
35          </ui:html>
36       </ui:page>
37    </f:view>
38 </jsp:root>
```

Figura 26.4 | Archivo JSP generado por Java Studio Creator 2, que muestra la hora actual en el servidor Web.

26.5.1 Análisis de un archivo JSP

Los archivos JSP que se utilizan en este ejemplo (y los siguientes) se generan casi completamente mediante Java Studio Creator 2, el cual proporciona un Editor visual que nos permite crear la GUI de una página al arrastrar y soltar componentes en un área de diseño. El IDE genera un archivo JSP en respuesta a las interacciones del programador. En la línea 1 de la figura 26.4 está la declaración XML, la cual indica que la JSP está expresada en sintaxis XML, junto con la versión de XML que se utiliza. En las líneas 3 a 5 hay comentarios que agregamos a la JSP, para indicar su número de figura, nombre de archivo y propósito.

En la línea 6 empieza el elemento raíz para la JSP. Todas las JSPs deben tener este elemento **jsp:root**, el cual tiene un atributo version para indicar la versión de JSP que se está utilizando (línea 6), y uno o más atributos xmlns (líneas 7 a 10). Cada **atributo xmlns** especifica un prefijo y un URL para una biblioteca de etiquetas, lo cual permite a la página usar las etiquetas especificadas en esa biblioteca. Por ejemplo, la línea 9 permite a la página usar los elementos estándar de las JSPs. Para usar estos elementos, hay que colocar el prefijo jsp antes de la etiqueta de cada elemento. Todas las JSPs generadas por Java Studio Creator 2 incluyen las bibliotecas de etiquetas especificadas en las líneas 7 a 10 (la biblioteca de componentes JSF básicos, la biblioteca de componentes JSF de HTML, la biblioteca de componentes JSP estándar y la biblioteca de componentes JSF de interfaz de usuario).

En las líneas 11 y 12 se encuentra el elemento **jsp:directive.page**. Su atributo contentType especifica el tipo MIME (text/html) y el conjunto de caracteres (UTF-8) que utiliza la página. El atributo pageEncoding especifica la codificación de caracteres que utiliza el origen de la página. Estos atributos ayudan al cliente (por lo general, un navegador Web) a determinar cómo desplegar el contenido.

Todas las páginas que contienen componentes JSF se representan en un **árbol de componentes** (figura 26.5) con el elemento JSF raíz **f:view**, que es de tipo **UIViewRoot**. Para representar la estructura de este árbol de componentes en una JSP, se encierran todas las etiquetas de los componentes JSF dentro del elemento f:view (líneas 13 a 37).

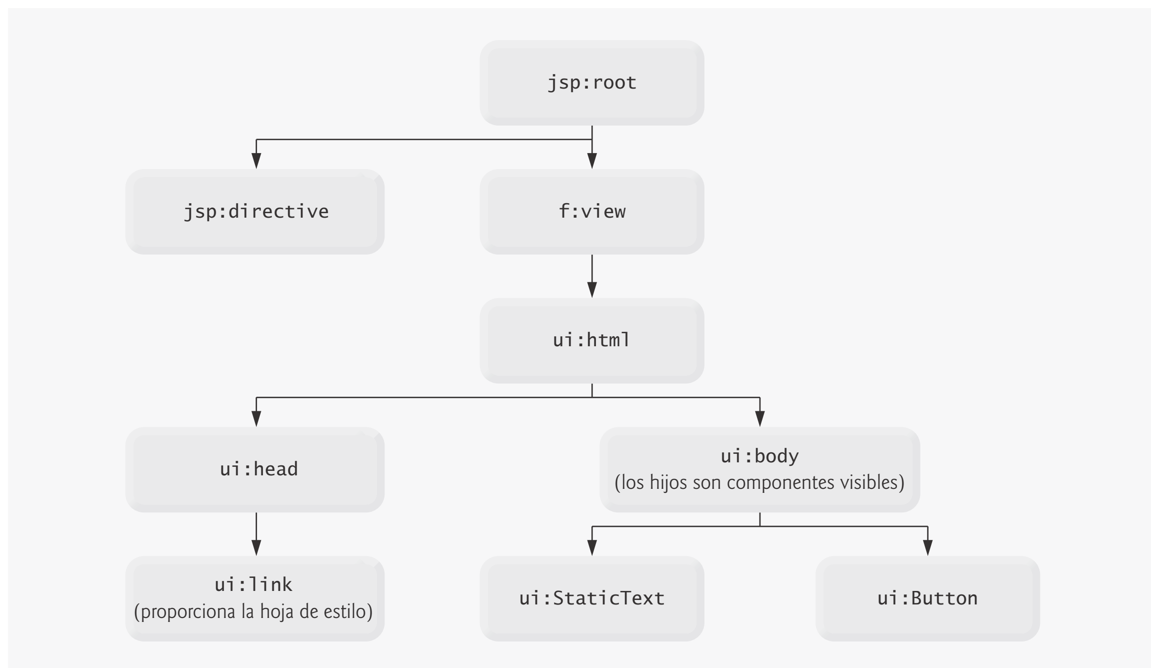


Figura 26.5 | Árbol de componentes JSF de ejemplo.

En las líneas 14 a 20 empieza la definición de la JSP con las etiquetas **ui:page**, **ui:html** y **ui:head**, todas de la biblioteca de etiquetas ui (componentes JSF de interfaz de usuario). Éstos y muchos otros elementos de página tienen un atributo binding. Por ejemplo, el elemento ui:head (línea 16) tiene el atributo binding = "#{Hora.head}.". Este atributo utiliza la notación del **Lenguaje de expresiones JSF** (es decir, #{Hora.head}) para hacer referencia a la propiedad head en la clase Hora que representa al bean de página (en la figura 26.6 podrá ver esta clase). Es posible enlazar un solo atributo de un elemento JSP a una propiedad en cualquiera de los JavaBeans de la aplicación Web. Por ejemplo, el atributo text de un componente **ui:label** se puede enlazar a una propiedad String en el objeto SessionBean de la aplicación. En la sección 26.7.2 veremos un ejemplo de esto.

El elemento **ui:head** (líneas 16 a 20) tiene un atributo title que especifica el título de la página. Este elemento también contiene un elemento **ui:link** (líneas 18 y 19), el cual especifica la hoja de estilo CSS que utiliza la página. El elemento **ui:body** (líneas 22 a 34) contiene un elemento **ui:form** (líneas 24 a 33), el cual contiene

dos componentes `ui:staticText` (líneas 25 a 28 y 29 a 32). Estos componentes muestran el texto de la página. El componente `encabezadoHora` (líneas 25 a 28) tiene un atributo `text` (líneas 27 y 28) que especifica el texto a mostrar (es decir, `"Hora actual en el servidor Web:"`). El componente `textoReloj` (líneas 29 a 32) no especifica un atributo de `texto`, ya que el texto de este componente se establecerá mediante programación.

Para que el marcado en este archivo se muestre en un navegador Web, todos los elementos de la JSP se asignan automáticamente a elementos de XHTML que el navegador reconoce. El mismo componente Web se puede asignar a varios elementos de XHTML distintos, dependiendo del navegador Web cliente y de las configuraciones de las propiedades del componente. En este ejemplo, los componentes `ui:staticText` (líneas 25 a 28, 29 a 32) se asignan a elementos **`span`** de XHTML. Un elemento `span` contiene texto que se muestra en una página Web, y que comúnmente se utiliza para controlar el formato del texto. Los atributos `style` de un elemento `ui:staticText` de una JSP se representan como parte del correspondiente atributo `style` del elemento `span` cuando el navegador despliega la página. En un momento le mostraremos el documento XHTML que se produce cuando un navegador solicita la página `Hora.jsp`.

26.5.2 Análisis de un archivo de bean de página

En la figura 26.6 se presenta el archivo de bean de página. En la línea 3 se indica que esta clase pertenece al paquete `horaweb`. Esta línea se genera automáticamente y especifica el nombre del proyecto como el nombre del paquete. En la línea 17 empieza la declaración de la clase `Hora` e indica que hereda de la clase **`AbstractPageBean`** (del paquete `com.sun.rave.web.ui.appbase`). Todas las clases de bean de página que soportan archivos JSP con componentes JSF deben heredar de la clase abstracta **`AbstractPageBean`**, la cual proporciona métodos para el ciclo de vida de las páginas. Observe que el IDE hace que el nombre de la clase coincida con el nombre de la página. El paquete **`com.sun.rave.web.ui.component`** incluye clases para muchos de los componentes JSF básicos (vea las instrucciones `import` en las líneas 6 a 11 y 13).

```
1  // Fig. 26.6: Hora.java
2  // Archivo de bean de página que establece textoReloj a la hora en el servidor Web.
3  package horaweb;
4
5  import com.sun.rave.web.ui.appbase.AbstractPageBean;
6  import com.sun.rave.web.ui.component.Body;
7  import com.sun.rave.web.ui.component.Form;
8  import com.sun.rave.web.ui.component.Head;
9  import com.sun.rave.web.ui.component.Html;
10 import com.sun.rave.web.ui.component.Link;
11 import com.sun.rave.web.ui.component.Page;
12 import javax.faces.FacesException;
13 import com.sun.rave.web.ui.component.StaticText;
14 import java.text.DateFormat;
15 import java.util.Date;
16
17 public class Hora extends AbstractPageBean
18 {
19    private int __placeholder;
20
21    // método de inicialización de componentes, generado automáticamente.
22    private void _init() throws Exception
23    {
24       // cuerpo vacío
25    } // fin del método _init
26
27    private Page page1 = new Page();
28
29    public Page getPage1()
30    {
```

Figura 26.6 | Archivo de bean de página que establece `textoReloj` a la hora en el servidor Web. (Parte 1 de 4).

```
         return page1;
      } // fin del método getPage1

      public void setPage1(Page p)
      {
         this.page1 = p;
      } // fin del método setPage1

      private Html html1 = new Html();

      public Html getHtml1()
      {
         return html1;
      } // fin del método getHtml1

      public void setHtml1(Html h)
      {
         this.html1 = h;
      } // fin del método setHtml1

      private Head head1 = new Head();

      public Head getHead1()
      {
         return head1;
      } // fin del método getHead1

      public void setHead1(Head h)
      {
         this.head1 = h;
      } // fin del método setHead1

      private Link link1 = new Link();

      public Link getLink1()
      {
         return link1;
      } // fin del método getLink1

      public void setLink1(Link l)
      {
         this.link1 = l;
      } // fin del método setLink1

      private Body body1 = new Body();

      public Body getBody1()
      {
         return body1;
      } // fin del método getBody1

      public void setBody1(Body b)
      {
         this.body1 = b;
      } // fin del método setBody1

      private Form form1 = new Form();

      public Form getForm1()
```

Figura 26.6 | Archivo de bean de página que establece `textoReloj` a la hora en el servidor Web. (Parte 2 de 4).

```
   {
      return form1;
   } // fin del método getForm1

   public void setForm1(Form f)
   {
      this.form1 = f;
   } // fin del método setForm1

   private StaticText encabezadoHora = new StaticText();

   public StaticText getEncabezadoHora()
   {
      return encabezadoHora;
   } // fin del método getEncabezadoHora

   public void setEncabezadoHora(StaticText st)
   {
      this.encabezadoHora = st;
   } // fin del método setEncabezadoHora

   private StaticText textoReloj = new StaticText();

   public StaticText getTextoReloj()
   {
      return textoReloj;
   } // fin del método getTextoReloj

   public void setTextoReloj(StaticText st)
   {
      this.textoReloj = st;
   } // fin del método setTextoReloj

   // Construye una nueva instancia de bean de página
      public Hora()
   {
      // constructor vacío
   } // fin del constructor

   // Devuelve una referencia al bean de datos con ámbito
   protected RequestBean1 getRequestBean1()
   {
      return (RequestBean1)getBean("RequestBean1");
   } // fin del método getRequestBean1

   // Devuelve una referencia al bean de datos con ámbito
   protected ApplicationBean1 getApplicationBean1()
   {
      return (ApplicationBean1)getBean("ApplicationBean1");
   } // fin del método getApplicationBean1

   // Devuelve una referencia al bean de datos con ámbito
   protected SessionBean1 getSessionBean1()
   {
      return (SessionBean1)getBean("SessionBean1");
   } // fin del método getSessionBean1

   // inicializa el contenido de la página
   public void init()
```

Figura 26.6 | Archivo de bean de página que establece `textoReloj` a la hora en el servidor Web. (Parte 3 de 4).

```
149      {
150         super.init();
151         try
152         {
153            _init();
154         } // fin de try
155         catch ( Exception e )
156         {
157            log( "Error al inicializar Hora", e );
158            throw e instanceof FacesException ? ( FacesException ) e:
159               new FacesException( e );
160         } // fin de catch
161      } // fin del método init
162
163      // método que se llama cuando ocurre una petición de devolución de envío
164      public void preprocess()
165      {
166         // cuerpo vacío
167      } // fin del método preprocess
168
169      // método al que se llama antes de desplegar la página
170      public void prerender()
171      {
172         textoReloj.setValue( DateFormat.getTimeInstance(
173            DateFormat.LONG ).format( new Date() ) );
174      } // fin del método prerender
175
176      // método al que se llama una vez que se completa el despliegue, si se llamó a init
177      public void destroy()
178      {
179         // cuerpo vacío
180      } // fin del método destroy
181   } // fin de la clase Hora
```

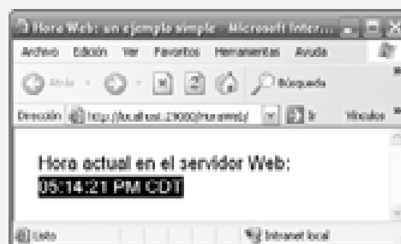


Figura 26.6 | Archivo de bean de página que establece `textoReloj` a la hora en el servidor Web. (Parte 4 de 4).

Este archivo de bean de página proporciona métodos *obtener (get)* y *establecer (set)* para cada elemento del archivo JSP de la figura 26.4. El IDE genera estos métodos de manera automática. Incluimos el archivo de bean de página completo en este primer ejemplo, pero en los siguientes ejemplos omitiremos estas propiedades y sus métodos *obtener* y *establecer* para ahorrar espacio. En las líneas 99 a 109 y 111 a 121 del archivo de bean de página se definen los dos componentes **Static Text** que soltamos en la página, junto con sus métodos *obtener* y *establecer*. Estos componentes son objetos de la clase `StaticText` en el paquete `com.sun.rave.web.ui.component`.

La única lógica requerida en esta página es establecer el texto del componente `textoReloj` para que lea la hora actual en el servidor. Esto lo hacemos en el método `prerender` (líneas 170 a 174). Más adelante hablaremos sobre el significado de éste y otros métodos de bean de página. En las líneas 172 y 173 se obtiene y da formato a la hora en el servidor, y se establece el valor de `textoReloj` con esa hora.

26.5.3 Ciclo de vida del procesamiento de eventos

El modelo de aplicación de Java Studio Creator 2 coloca varios métodos en el bean de página, los cuales se enlazan en el **ciclo de vida del procesamiento de eventos**. Estos métodos representan cuatro etapas principales: inicialización, pre-procesamiento, pre-despliegue y destrucción. Cada uno de ellos corresponde a un método en la clase de bean de página: `init`, `preprocess`, `prerender` y `destroy`, respectivamente. Java Studio Creator 2 crea estos métodos de manera automática, pero podemos personalizarlos para manejar las tareas de procesamiento del ciclo de vida, como desplegar un elemento en una página sólo si un usuario hace clic en un botón.

El **método `init`** (figura 26.6, líneas 148 a 161) es llamado por el contenedor de JSPs la primera vez que se solicita la página, y en las peticiones de devolución de envío. Una **petición de devolución de envío (postback)** ocurre cuando se envían los datos de un formulario, y la página junto con su contenido se envían al servidor para ser procesados. El método `init` invoca la versión de su superclase (línea 150) y después trata de llamar al método `_init` (declarado en las líneas 22 a 25). El método `_init` también se genera en forma automática, y maneja las tareas de inicialización de componentes (si los hay), como establecer las opciones para un grupo de botones de opción.

El **método `preprocess`** (líneas 164 a 167) se llama después de `init`, pero sólo si la página está procesando una petición de devolución de envío. El **método `prerender`** (líneas 170 a 174) se llama justo antes de que el navegador despliegue (muestre) una página. Este método se debe utilizar para establecer las propiedades de los componentes; las propiedades que se establecen antes (como en el método `init`) pueden sobrescribirse antes de que el navegador despliegue la página. Por esta razón, establecemos el valor de `textoReloj` en el método `prerender`.

Por último, el **método `destroy`** (líneas 177 a 180) se llama una vez que la página se ha desplegado, pero sólo si se hizo la llamada al método `init`. Este método maneja tareas tales como liberar los recursos que se utilizan para desplegar la página.

26.5.4 Relación entre la JSP y los archivos de bean de página

El bean de página tiene una propiedad para cada elemento que aparece en el archivo JSP de la figura 26.4, desde el elemento `html` hasta los dos componentes `Texto estático`. Recuerde que los elementos en el archivo JSP se enlazaron explícitamente a estas propiedades mediante el atributo `binding` de cada elemento, usando una instrucción en Lenguaje de expresiones JSF. Como ésta es una clase JavaBean, también se incluyen métodos *obtener (get)* y *establecer (set)* para cada una de estas propiedades (líneas 27 a 121). El IDE genera este código automáticamente para cada proyecto de aplicación Web.

26.5.5 Análisis del XHTML generado por una aplicación Web de Java

En la figura 26.7 se muestra el XHTML que se genera cuando un navegador Web cliente solicita la página `Hora.jsp` (figura 26.4). Para ver este XHTML, seleccione **Ver > Código fuente** en Internet Explorer. [*Nota:* agregamos los comentarios de XHTML en las líneas 3 y 4, y cambiamos el formato del XHTML para que se conforme a nuestras convenciones de codificación].

El documento XHTML en la figura 26.7 es similar en estructura al archivo JSP de la figura 26.4. En las líneas 5 y 6 está la declaración del tipo de documento, la cual lo declara como documento `XHTML 1.0 Transicional`. Las etiquetas `ui:meta` en las líneas 9 a 13 son equivalentes a los encabezados HTTP, y se utilizan para controlar el comportamiento del navegador Web.

```
1  <?xml version = "1.0"?>
2
3  <!-- Fig. 26.7: Hora.html -->
4  <!—La respuesta XHTML generada cuando el navegador solicita el archivo Hora.jsp. -->
5  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
6     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
7  <html xmlns = "http://www.w3.org/1999/xhtml">
8     <head>
9        <meta content = "no-cache" http-equiv = "Pragma" />
```

Figura 26.7 | Respuesta XHTML generada cuando el navegador solicita el archivo `Hora.jsp`. (Parte 1 de 2).

```
10        <meta content = "no-cache" http-equiv = "Cache-Control" />
11        <meta content = "no-store" http-equiv = "Cache-Control" />
12        <meta content = "max-age=0" http-equiv = "Cache-Control" />
13        <meta content = "1" http-equiv = "Expires" />
14        <title>Hora Web: un ejemplo simple</title>
15        <script type = "text/javascript"
16           src = "/HoraWeb/theme/com/sun/rave/web/ui/defaulttheme/
17           javascript/formElements.js"></script>
18        <link rel = "stylesheet" type = "text/css" href = "/HoraWeb/theme/
19           com/sun/rave/web/ui/defaulttheme/css/css_master.css" />
20        <link rel = "stylesheet" type = "text/css" href = "/HoraWeb/theme/
21           com/sun/rave/web/ui/defaulttheme/css/css_ie55up.css" />
22        <script type = "text/javascript">
23           var sjwuic_ScrollCookie = new sjwuic_ScrollCookie(
24           '/Hora.jsp', '/HoraWeb/faces/Hora.jsp' );
25        </script>
26        <link id = "link1" rel = "stylesheet" type = "text/css"
27           href = "/HoraWeb/resources/stylesheet.css" />
28     </head>
29     <meta id = "_id0" http-equiv = "refresh" content = "5" />
30     <body id = "body1" style = "-rave-layout: grid">
31     <form id = "form1" class = "form" method = "post"
32        action = "/HoraWeb/faces/Hora.jsp"
33        enctype = "application/x-www-form-urlencoded">
34        <span id = "form1:encabezadoHora" style = "font-size: 18px; left: 24px;
35           top: 24px; position: absolute">Hora actual en el servidor Web:
36        </span>
37        <span id = "form1:textoReloj" style = "background-color: black;
38           color: yellow; font-size: 18px; left: 24px; top:48px; position:
39           absolute">10:28:47 PM CDT</span>
40        <input id = "form1_hidden" name = "form1_hidden"
41           value = "form1_hidden" type = "hidden" />
42     </form>
43     </body>
44  </html>
```

Figura 26.7 | Respuesta XHTML generada cuando el navegador solicita el archivo `Hora.jsp`. (Parte 2 de 2).

En las líneas 30 a 43 se define el cuerpo (`body`) del documento. En la línea 31 empieza el formulario (`form`), un mecanismo para recolectar información del usuario y enviarla de vuelta al servidor Web. En este programa específico, el usuario no envía datos al servidor Web para procesarlos; sin embargo, el procesamiento de los datos del usuario es una parte imprescindible de muchas aplicaciones Web, la cual se facilita mediante el uso de los formularios. En ejemplos posteriores demostraremos cómo enviar datos al servidor.

Los formularios XHTML pueden contener componentes visuales y no visuales. Los componentes visuales incluyen botones y demás componentes de GUI con los que interactúan los usuarios. Los componentes no visuales, llamados **elementos de formulario `hidden`**, almacenan datos tales como direcciones de e-mail, que el autor del documento especifica. Una de estas entradas ocultas se define en las líneas 40 y 41. Más adelante en este capítulo hablaremos sobre el significado preciso de esta entrada oculta. El atributo **`method`** del elemento `form` (línea 31) especifica el método mediante el cual el navegador Web envía el formulario al servidor. De manera predeterminada, las JSPs utilizan el método `post`. Los dos **tipos de peticiones HTTP** más comunes (también conocidas como **métodos de petición**) son **`get`** y **`post`**. Una petición `get` obtiene (o recupera) la información de un servidor. Dichas peticiones comúnmente recuperan un documento HTML o una imagen. Una petición `post` envía datos a un servidor, como la información de autenticación o los datos de un formulario que recopilan la entrada del usuario. Por lo general, las peticiones `post` se utilizan para enviar un mensaje a un grupo de noticias o a un foro de discusión, pasar la entrada del usuario a un proceso manejador de datos en el servidor, y almacenar o actualizar los datos en un servidor. El atributo **`action`** de `form` (línea 32) identifica el recurso que se pedirá cuando se envíe este formulario; en este caso, `/HoraWeb/faces/Hora.jsp`.

Observe que los dos componentes **Texto estático** (es decir, `encabezadoHora` y `textoReloj`) se representan mediante dos elementos `span` en el documento XHTML (líneas 34 a 36, 37 a 39) como vimos anteriormente. Las opciones de formato que se especificaron como propiedades de `encabezadoHora` y `textoReloj`, como el tamaño de la fuente y el color del texto en los componentes, ahora se especifican en el atributo `style` de cada elemento `span`.

26.5.6 Creación de una aplicación Web en Java Studio Creator 2

Ahora que hemos presentado el archivo JSP, el archivo de bean de página y la página Web de XHTML resultante que se envía al navegador Web, vamos a describir los pasos para crear esta aplicación. Para crear la aplicación `HoraWeb`, realice los siguientes pasos en Java Studio Creator 2:

Paso 1: Creación del proyecto de aplicación Web

Seleccione **Archivo > Nuevo proyecto...** para mostrar el cuadro de diálogo **Nuevo proyecto**. En este cuadro de diálogo, seleccione **Web** en el panel **Categorías**, **Aplicación Web JSF** en el panel **Proyectos** y haga clic en **Siguiente**. Cambie el nombre del proyecto a `HoraWeb` y use la ubicación predeterminada del proyecto y el paquete Java predeterminado. Estas opciones crearán un directorio `HoraWeb` en su directorio `Mis documentos\Creator\Projects` para almacenar los archivos del proyecto. Haga clic en **Terminar** para crear el proyecto de aplicación Web.

Paso 2: Análisis de la ventana del Editor visual del nuevo proyecto

Las siguientes figuras describen características importantes del IDE, empezando con la ventana **Editor visual** (figura 26.8). Java Studio Creator 2 crea una sola página Web llamada `Page1` cuando se crea un nuevo proyecto. Esta página se abre de manera predeterminada en el Editor visual en **modo Diseño**, cuando el proyecto se carga por primera vez. A medida que arrastre y suelte nuevos componentes en la página, el modo **Diseño** le permitirá ver cómo se desplegará su página en el navegador. El archivo JSP para esta página, llamado `Page1.jsp`, se puede ver haciendo clic en el botón **JSP** que se encuentra en la parte superior del Editor visual, o haciendo clic con el botón derecho del ratón en cualquier parte dentro del Editor visual y seleccionando la opción **Editar origen JSP**. Como dijimos antes, cada página Web está soportada por un archivo de bean de página. Java Studio Creator 2 crea un archivo llamado `Page1.java` cuando se crea un nuevo proyecto. Para abrir este archivo, haga clic en el botón **Java** que se encuentra en la parte superior del Editor visual, o haga clic con el botón derecho del ratón en cualquier parte dentro del Editor visual y seleccione la opción **Editar origen Java Page1**.

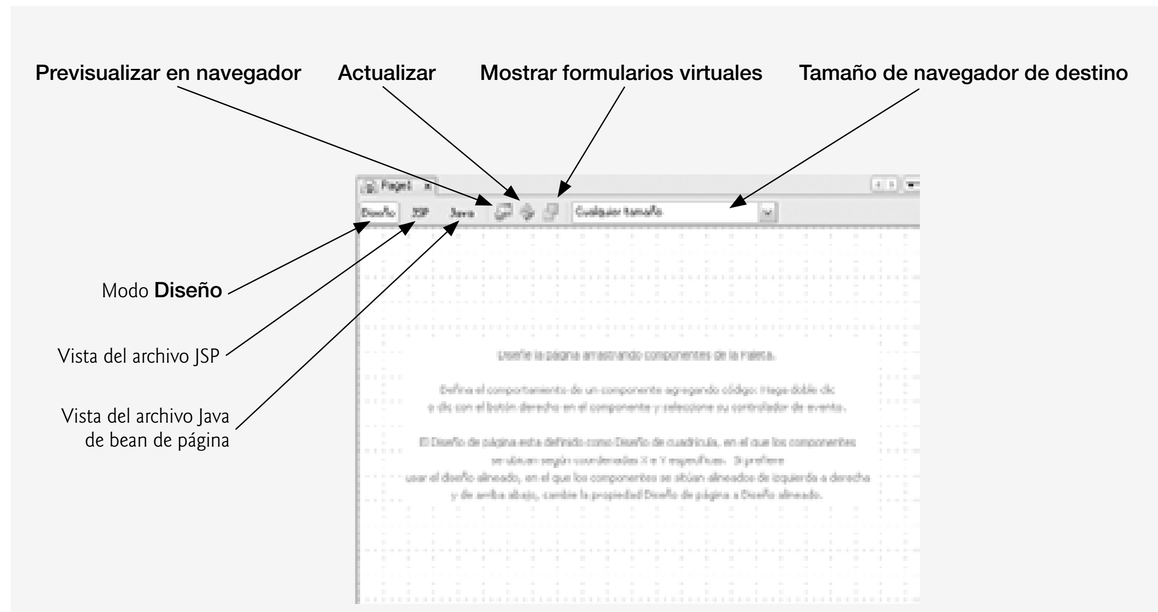


Figura 26.8 | Ventana Editor visual en modo **Diseño**.

El botón **Previsualizar en navegador** en la parte superior de la ventana Editor visual le permite ver sus páginas en un navegador sin tener que crear y ejecutar la aplicación. El botón **Actualizar** vuelve a dibujar la página en el Editor visual. El botón **Mostrar formularios virtuales** nos permite ver qué elementos de formulario están participando en los formularios virtuales (hablaremos sobre este concepto en el capítulo 27). La lista desplegable **Tamaño de navegador de destino** nos permite especificar la resolución óptima del navegador para ver la página, y nos permite ver cuál será la apariencia de la página en distintas resoluciones de pantalla.

Paso 3: Análisis de la Paleta *en Java Studio Creator 2*

En la figura 26.9 se muestra la **Paleta** que aparece en el IDE cuando se carga el proyecto. La parte a) muestra el inicio de la lista **Básicos** de componentes Web, y la parte b) muestra el resto de los componentes **Básicos**, junto con la lista de componentes **Diseño**. Hablaremos sobre componentes específicos de la figura 26.9 a medida que los utilicemos en el capítulo.

Paso 4: Análisis de la ventana Proyectos

En la figura 26.10 se muestra la ventana **Proyectos**, la cual aparece en la esquina inferior derecha del IDE. Esta ventana muestra la jerarquía de todos los archivos incluidos en el proyecto. Los archivos JSP para cada página se enlistan en el nodo **Páginas Web**. Este nodo también incluye la carpeta **resources**, la cual contiene la hoja de estilo CSS para el proyecto, y cualquier otro archivo que puedan necesitar las páginas para mostrarse en forma apropiada, como los archivos de imagen. Todo el código fuente de Java, incluyendo el archivo de bean de página para cada página Web y los beans de aplicación, sesión y petición, se pueden encontrar bajo el nodo **Paquetes de origen**. Otro archivo útil que se muestra en la ventana del proyecto es el archivo **Navegación de página**, el cual define las reglas para navegar por las páginas del proyecto, con base en el resultado de algún evento iniciado por el usuario, como hacer clic en un botón o en un vínculo. También podemos acceder al archivo **Navegación de página** si hacemos clic con el botón derecho del ratón en el Editor visual, estando en modo **Diseño**; para ello, seleccionamos la opción **Navegación de página...**.

Paso 5: Análisis de los archivos JSP y Java en el IDE

En la figura 26.11 se muestra `Page1.jsp`; el archivo JSP generado por Java Studio Creator 2 para `Page1`. [*Nota:* cambiamos el formato del código para adaptarlo a nuestras convenciones de codificación]. Haga clic en el botón **JSP** que está en la parte superior del Editor visual para abrir el archivo JSP. Cuando se crea por primera vez, este

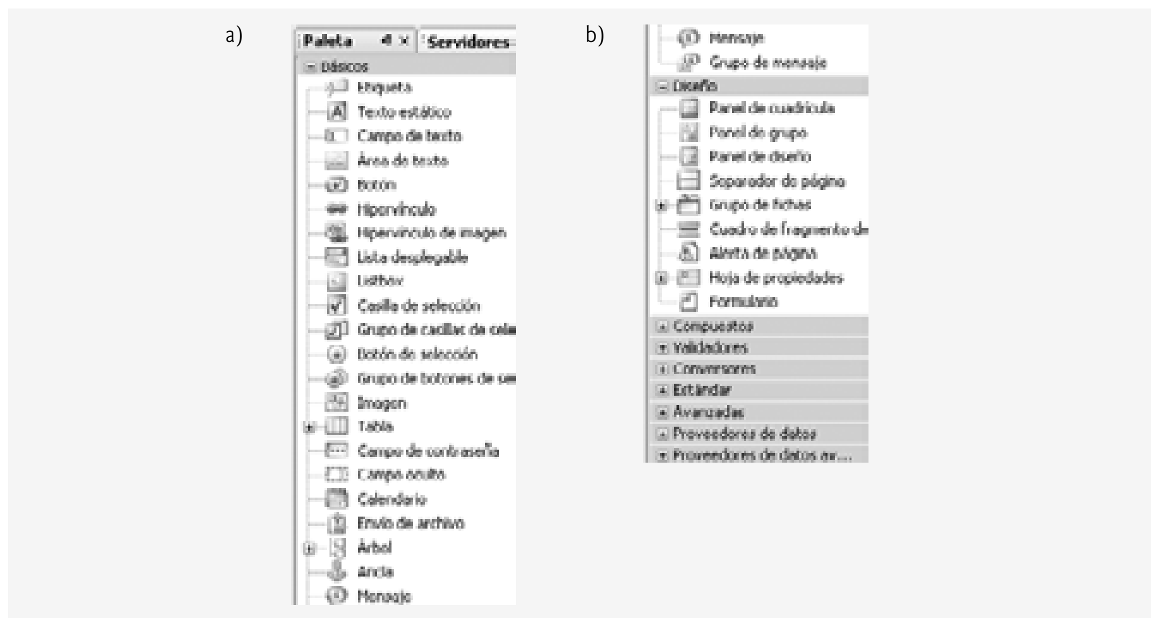


Figura 26.9 | La **Paleta** en Java Studio Creator 2.

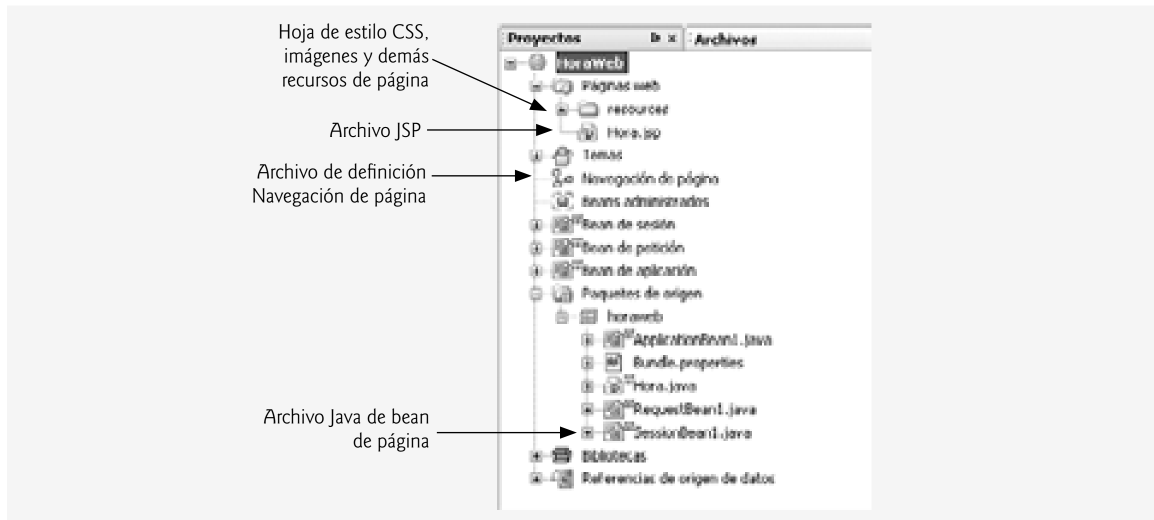


Figura 26.10 | Ventana **Proyectos** para el proyecto `HoraWeb`.

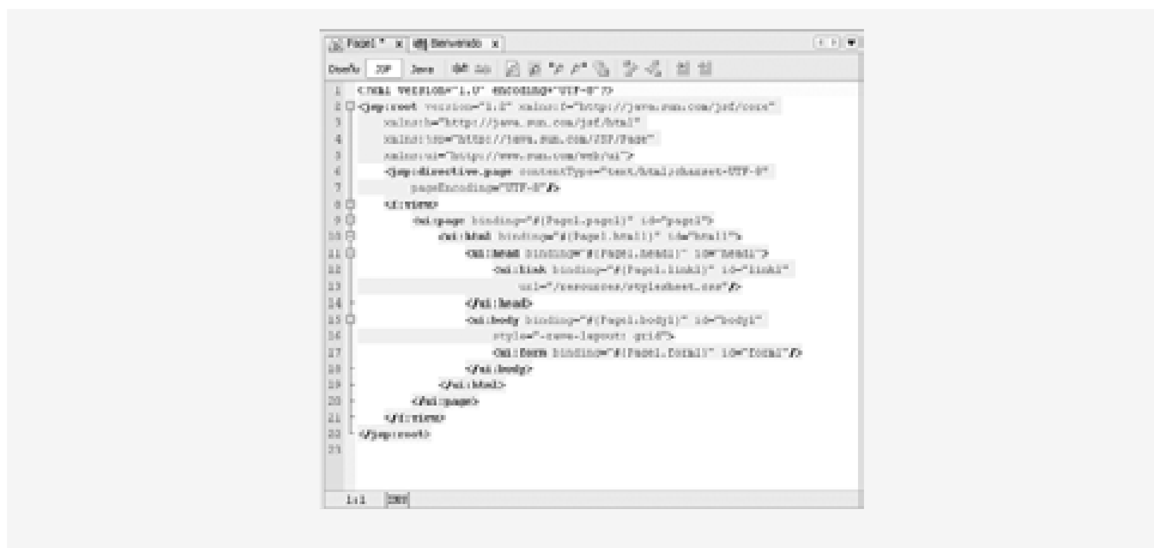


Figura 26.11 | Archivo JSP generado para la página 1 (Page1) por Java Studio Creator 2.

archivo contiene varias etiquetas para configurar la página, incluyendo creación de vínculos a la hoja de estilo de la página y definir las bibliotecas JSF necesarias. En cualquier otro caso, las etiquetas del archivo JSP están vacías, ya que no se han agregado todavía componentes a la página.

En la figura 26.12 se muestra parte de `Page1.java`; el archivo de bean de página generado por Java Studio Creator 2 para `Page1`. Haga clic en el botón **Java** que está en la parte superior del Editor visual para abrir el archivo de bean de página. Este archivo contiene una clase de Java con el mismo nombre que la página (es decir, `Page1`), la cual extiende a la clase `AbstractPageBean`. Como dijimos antes, `AbstractPageBean` tiene varios métodos para manejar el ciclo de vida de la página. Cuatro de estos métodos (`init`, `preprocess`, `prerender` y `destroy`) son sobrescritos por `Page1.java`. Excepto por el método `init`, estos métodos están vacíos al principio. Sirven como receptáculos para que el programador pueda personalizar el comportamiento de su aplicación Web. El archivo de bean de página también incluye métodos *establecer (set)* y *obtener (get)* para todos los elementos de la página: `page`, `html`, `head`, `body` y `link` para empezar. Para ver estos métodos *obtener* y *establecer*, haga clic en el signo más (+) en la línea que dice **Creator-managed Component Definition**.

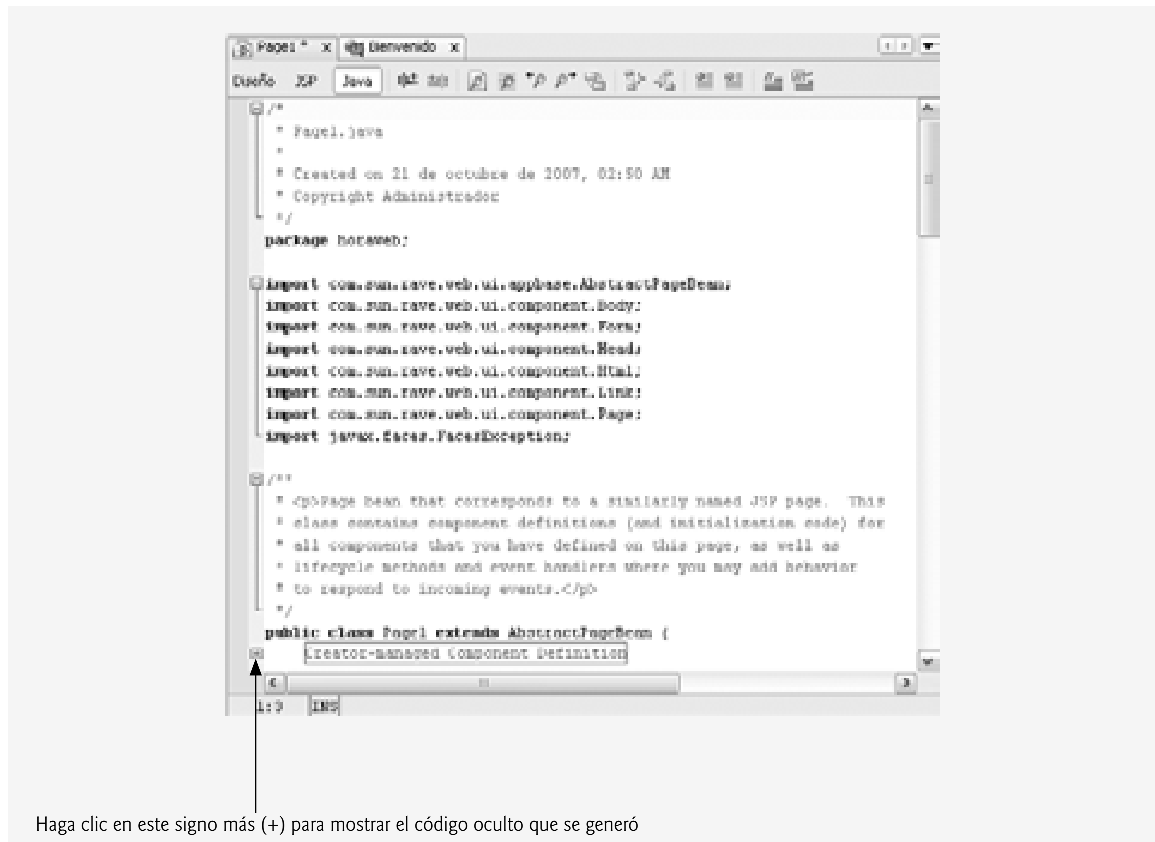


Figura 26.12 | Archivo de bean de página para `Page1.jsp`, generado por Java Studio Creator 2.

Paso 6: Cambiar el nombre de los archivos JSP y JSF

Por lo general, es conveniente cambiar el nombre de los archivos JSP y Java en un proyecto, de manera que sus nombres sean relevantes para nuestra aplicación. Haga clic con el botón derecho del ratón en el archivo `Page1.jsp` en la **Ventana Proyectos** y seleccione **Cambiar nombre,** para que aparezca el cuadro de diálogo **Cambiar nombre.** Escriba el nuevo nombre `Hora` para el archivo. Si está activada la opción **Previsualizar todos los cambios**, aparecerá la **Ventana Refactorización** en la parte inferior del IDE cuando haga clic en **Siguiente >**. La **Refactorización** es el proceso de modificar el código fuente para mejorar su legibilidad y reutilización, sin modificar su comportamiento; por ejemplo, al cambiar los nombres a los métodos o variables, o al dividir métodos extensos en varios métodos más cortos, Java Studio Creator 2 tiene herramientas de refactorización integradas, las cuales automatizan ciertas tareas de refactorización. Al usar estas herramientas para cambiar el nombre a los archivos del proyecto, se actualizan los nombres tanto del archivo JSP como del archivo de bean de página. La herramienta de refactorización también modifica el nombre de la clase en el archivo de bean de página y todos los enlaces de los atributos en el archivo JSP, para reflejar el nuevo nombre de la clase. Observe que no se hará ninguno de estos cambios, sino hasta que haga clic en el botón **Refactorizar** de la **Ventana Refactorización.** Si no previsualiza los cambios, la refactorización ocurre al momento en que haga clic en el botón **Siguiente >** del cuadro de diálogo **Cambiar nombre.**

Paso 7: Cambiar el título de la página

Antes de diseñar el contenido de la página Web, vamos a darle el título `"Hora Web: un ejemplo simple"`. De manera predeterminada, la página no tiene un título cuando el IDE la genera. Para agregar un título, abra el archivo JSP en modo **Diseño**. En la ventana **Propiedades**, escriba el nuevo título enseguida de la propiedad **Title** y oprima *Intro*. Vea la JSP para cerciorarse que el atributo `title = "Hora Web: un ejemplo simple"` se haya agregado automáticamente a la etiqueta `ui:head`.

Paso 8: Diseñar la página

Diseñar una página Web es más sencillo en Java Studio Creator 2. Para agregar componentes a la página, puede arrastrarlos y soltarlos desde la **Paleta** hacia la página en modo **Diseño**. Al igual que la misma página Web, cada componente es un objeto que tiene propiedades, métodos y eventos. Puede establecer estas propiedades y eventos en forma visual, mediante la ventana **Propiedades**, o mediante programación en el archivo de bean de página. Los métodos *obtener (get)* y *establecer (set)* se agregan automáticamente al archivo de bean de página para cada componente que se agregue a la página.

El IDE genera las etiquetas JSP para los componentes que el programador arrastra y suelta mediante el uso de un diseño de cuadrícula, como se especifica en la etiqueta `ui:body`. Esto significa que los componentes se desplegarán en el navegador usando **posicionamiento absoluto**, de manera que aparezcan exactamente en donde se sueltan en la página. A medida que el programador agregue componentes a la página, el atributo `style` en el elemento JSP de cada componente incluirá el número de píxeles desde los márgenes superior e izquierdo de la página en la que se posicione el componente.

En este ejemplo, usamos dos componentes **Texto estático**. Para agregar el primero a la página Web, arrástrelo y suéltelo desde la lista de componentes **Básicos** de la **Paleta**, hasta la página en modo **Diseño**. Edite el texto del componente, escribiendo `"Hora actual en el servidor Web:"` directamente en el componente. También puede editar el texto modificando la propiedad `text` del componente en la ventana **Propiedades**. Java Studio Creator 2 es un editor **WYSIWYG** (**What You See Is What You Get** —Lo que ve es lo que obtiene); cada vez que realice un cambio a una página Web en modo **Diseño**, el IDE creará el marcado (visible en modo **JSP**) necesario para lograr los efectos visuales deseados que aparecen en modo **Diseño**. Después de agregar el texto a la página Web, cambie al modo **JSP**. Ahí podrá ver que el IDE agregó un elemento `ui:staticText` al cuerpo de la página, el cual está enlazado al objeto `staticText1` en el archivo de bean de página, y cuyo atributo `text` coincide con el texto que acaba de escribir. De vuelta al modo **Diseño**, haga clic en el componente **Texto estático** para seleccionarlo. En la ventana **Propiedades**, haga clic en el botón de elipsis enseguida de la propiedad `style` para abrir un cuadro de diálogo y editar el estilo del texto. Seleccione `18 px` para el tamaño de la fuente y haga clic en **Aceptar**. De nuevo en la ventana **Propiedades**, cambie la propiedad `id` a `encabezadoHora`. Al establecer la propiedad `id` también se modifica el nombre de la propiedad correspondiente del componente en el bean de página, y se actualiza su atributo `binding` en la JSP de manera acorde. Observe que se ha agregado `font-size: 18 px` al atributo style y que el atributo `id` ha cambiado a `encabezadoHora` en la etiqueta del componente en el archivo JSP. El IDE debe aparecer ahora como se muestra en la figura 26.13.

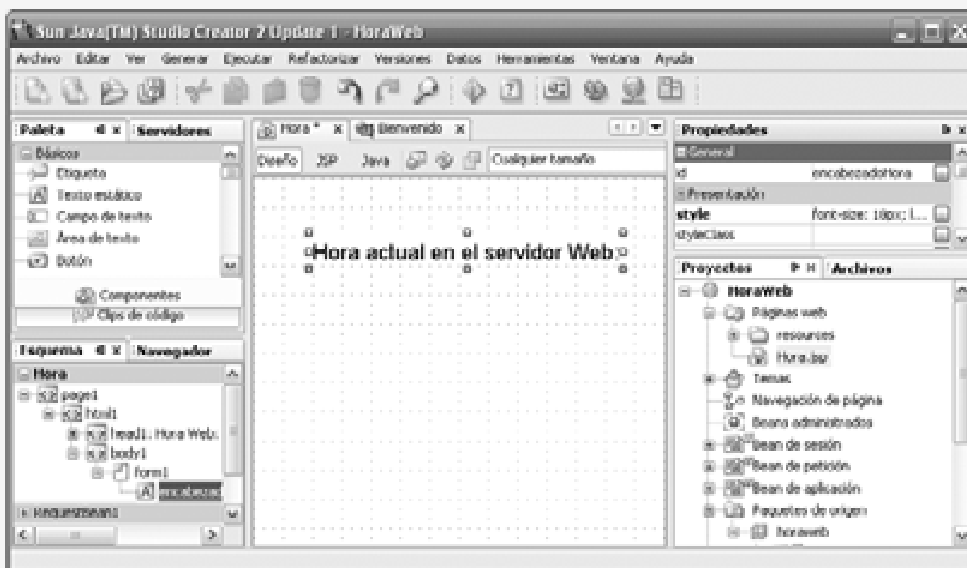


Figura 26.13 | `Hora.jsp` después de insertar el primer componente **Texto estático**.

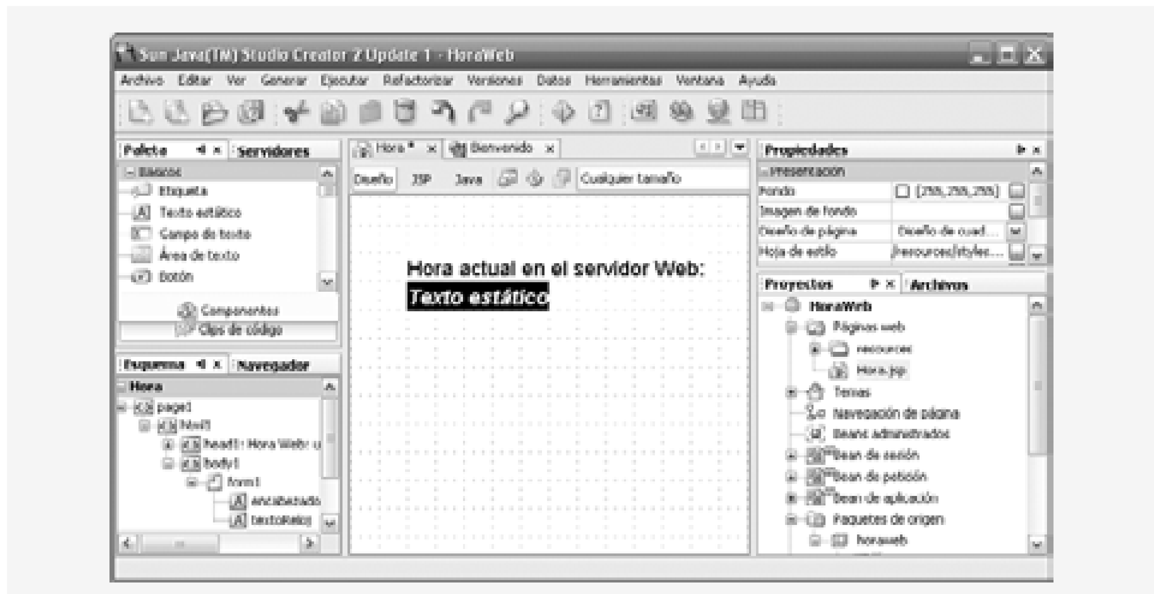


Figura 26.14 | `Hora.jsp` después de agregar el segundo componente `Texto estático`.

Coloque un segundo componente **Texto estático** en la página, y establezca su `id` en `textoReloj`. Edite su propiedad `style` de manera que el tamaño de la fuente sea `18 px`, el color de texto sea amarillo (`yellow`) y el color de fondo sea negro (`black`). No edite el texto del componente, ya que éste se establecerá mediante programación en el archivo de bean de página. El componente se mostrará con el texto ***Texto estático*** en el IDE, pero no mostrará ningún texto en tiempo de ejecución, a menos que éste se establezca mediante programación. La figura 26.14 muestra el IDE después de agregar el segundo componente.

Paso 9: Agregar la lógica de la página

Después de diseñar la interfaz de usuario, puede modificar el archivo de bean de página para establecer el texto del elemento `textoReloj`. En este ejemplo, agregamos una instrucción al método `prerender` (líneas 170 a 174 de la figura 26.6). Recuerde que utilizamos el método `prerender` para asegurar que `textoReloj` se actualice cada vez que se actualice la página. En las líneas 172 y 173 de la figura 26.6 se establece mediante programación el texto de `textoReloj` con la hora actual en el servidor.

Nos gustaría que esta página se actualizara automáticamente para mostrar una hora actualizada. Para lograr esto, agregue la etiqueta vacía `<ui: meta content = "60" httpEquiv = "refresh" />` al archivo JSP, entre el final de la etiqueta `ui:head` y el inicio de la etiqueta `ui:body`. Esta etiqueta indica al navegador que debe volver a cargar la página automáticamente cada 60 segundos. También puede agregar esta etiqueta arrastrando un componente **Meta** de la sección **Avanzados** de la **Paleta** a su página, y después estableciendo el atributo `content` del componente a `60` y su atributo `httpEquiv` a `refresh`.

Paso 10: Análisis de la ventana Esquema

En la figura 26.15 se muestra la **ventana Esquema** en Java Studio Creator 2. Los cuatro archivos Java del proyecto se muestran como nodos de color gris. El nodo **Hora** que representa el archivo de bean de página está expandido y muestra el contenido del árbol de componentes. Los beans de ámbito de petición, sesión y aplicación están contraídos de manera predeterminada, ya que no hemos agregado propiedades a estos beans en este ejemplo. Al hacer clic en el árbol de componentes de la página, se selecciona el elemento en el Editor visual.

Paso 11: Ejecutar la aplicación

Después de crear la página Web, podemos verla de varias formas. Primero seleccione **Generar > Generar proyecto principal**, y después que se complete la generación, seleccione **Ejecutar > Ejecutar proyecto principal** para ejecu-

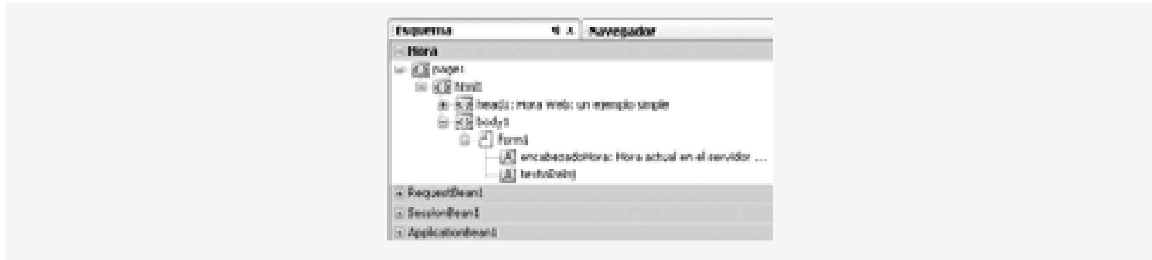


Figura 26.15 | Ventana **Esquema** en Java Studio Creator 2.

tar la aplicación en una ventana del navegador. Para ejecutar un proyecto que ya haya sido generado, oprima el icono **Ejecutar proyecto principal** ( ) en la barra de herramientas que se encuentra en la parte superior del IDE. Observe que, si se hacen cambios a un proyecto, éste debe volver a generarse para que puedan reflejarse cuando se vea la aplicación en un navegador Web. Como esta aplicación se generó en el sistema de archivos local, el URL que se muestre en la barra de dirección del navegador cuando se ejecute la aplicación será `http://localhost:29080/HoraWeb/` (figura 26.6), en donde 29080 es el número de puerto en el que se ejecuta el servidor de prueba integrado de Java Studio Creator 2 **(Sun Application Server 8)** de manera predeterminada. Al ejecutar un programa en el servidor de prueba, aparece un icono cerca de la parte inferior derecha de la pantalla, para demostrar que Sun Application Server se está ejecutando. Para cerrar el servidor después de salir de Java Studio Creator 2, haga clic con el botón derecho en el icono de la bandeja y seleccione **Stop Domain creator**.

De manera alternativa, puede oprimir *F5* para generar la aplicación y después ejecutarla en modo de depuración; el depurador integrado de Java Studio Creator 2 puede ayudarle a diagnosticar fallas en las aplicaciones. Si escribe <*Ctrl*> *F5*, el programa se ejecuta sin habilitar la depuración.



Tip para prevenir errores 26.1

Si tiene problemas al generar su proyecto debido a errores en los archivos XML generados por Java Studio Creator, que se utilizan para la generación, pruebe a limpiar el proyecto y volver a generar. Para ello, seleccione ***Generar > Limpiar y generar proyecto principal****, u oprima* <Alt> B.

Por último, para ejecutar su aplicación generada, abra una ventana del navegador y escriba el URL de la página Web en el campo **Dirección**. Como su aplicación reside en el sistema de archivos local, primero debe iniciar Sun Application Server. Si ejecutó antes la aplicación utilizando uno de los métodos anteriores, el servidor ya se estará ejecutando. De no ser así, puede iniciar el servidor desde el IDE; para ello abra la ficha **Servidores** (que se encuentra en el mismo panel que la **Paleta**), haga clic con el botón derecho del ratón en el **Servidor de ejecución**, seleccione **Iniciar/Detener servidor** y haga clic en el botón **Iniciar**, en el cuadro de diálogo que aparezca. Después, puede escribir el URL (incluyendo el número de puerto para el servidor de aplicación, 29080) en el navegador para ejecutar la aplicación. Para este ejemplo no es necesario escribir el URL completo, `http://localhost:29080/HoraWeb/faces/Hora.jsp`. La ruta para el archivo `Hora.jsp` (es decir, `faces/Hora.jsp`) se puede omitir, ya que este archivo se estableció de manera predeterminada como la página inicial del proyecto. Para los proyectos con varias páginas, puede modificar la página inicial haciendo clic en la página deseada en la ventana **Proyectos**, y seleccionando **Definir como página de inicio**. La página de inicio se indica mediante una flecha verde enseguida del nombre de la página en la ventana **Proyectos**. [*Nota:* si utiliza Netbeans Visual Web Pack 5.5, el número de puerto dependerá del servidor en el que despliegue su aplicación Web. Además, la ficha **Servidores** se llama **Tiempo de ejecución** (**Runtime**) en Netbeans].

26.6 Componentes JSF

En esta sección presentaremos algunos de los componentes JSF que se incluyen en la **Paleta** (figura 26.9). En la figura 26.16 se sintetizan algunos de los componentes JSF que se utilizan en los ejemplos del capítulo.

26.6.1 Componentes de texto y gráficos

En la figura 26.17 se muestra un formulario simple para recopilar la entrada del usuario. Este ejemplo utiliza todos los componentes enlistados en la figura 26.16, con la excepción de `Etiqueta`, que veremos en ejemplos

Componentes JSF	Descripción
Etiqueta	Muestra texto que se puede asociar con un elemento de entrada.
Texto estático	Muestra texto que el usuario no puede editar.
Campo de texto	Recopila la entrada del usuario y muestra texto.
Botón	Desencadena un evento cuando se oprime.
Hipervínculo	Muestra un hipervínculo.
Lista desplegable	Muestra una lista desplegable de opciones.
Grupo de botones de selección	Muestra botones de opción.
Imagen	Muestra imágenes (como GIF y JPG).

Figura 26.16 | Componentes JSF de uso común.

posteriores. Todo el código en la figura 26.17 se generó mediante Java Studio Creator 2, en respuesta a las acciones realizadas en modo **Diseño**. Este ejemplo no realiza ninguna tarea cuando el usuario hace clic en **Registrar**. Le pediremos que agregue funcionalidad a este ejemplo como un ejercicio. En los siguientes ejemplos, demostraremos cómo agregar funcionalidad a muchos de estos componentes JSF.

Antes de hablar sobre los componentes JSF que se utilizan en este archivo JSP, explicaremos el XHTML que crea el esquema de la figura 26.17. Como dijimos antes, Java Studio Creator 2 utiliza el posicionamiento absoluto, por lo que los componentes se despliegan en donde se hayan soltado en el Editor visual. En este ejemplo, además del posicionamiento absoluto utilizamos un componente `Panel de cuadrícula` (líneas 31 a 52) del grupo de componentes **Diseño** de la **Paleta**. El prefijo `h:` indica que se encuentra en la biblioteca de etiquetas HTML de JSF. Este componente, un objeto de la clase `HtmlPanelGrid` en el paquete `javax.faces.component.html`, controla el posicionamiento de los componentes que contiene. El componente **Panel de cuadrícula** permite al diseñador especificar el número de columnas que debe contener la cuadrícula. Después se pueden soltar los componentes en cualquier parte dentro del panel, y éstos se reposicionarán automáticamente en columnas espaciadas de manera uniforme, en el orden en el que se suelten. Cuando el número de componentes excede al número de columnas, el panel desplaza los componentes adicionales hacia una nueva fila. De esta forma, el **Panel de cuadrícula** se comporta como una tabla de XHTML, y de hecho se despliega en el navegador como una tabla XHTML. En este ejemplo, usamos el **Panel de cuadrícula** para controlar las posiciones de los componentes **Imagen** y **Campo de texto** en la sección de la página acerca de la información del usuario.

```
1  <?xml version = "1.0" encoding = "UTF-8"?>
2
3  <!-- Fig. 26.17: ComponentesWeb.jsp -->
4  <!-- Formulario de registro que demuestra el uso de los componentes JSF. -->
5  <jsp:root version = "1.2" xmlns:f = "http://java.sun.com/jsf/core"
6     xmlns:h = "http://java.sun.com/jsf/html" xmlns:jsp =
7     "http://java.sun.com/JSP/Page" xmlns:ui = "http://www.sun.com/web/ui">
8     <jsp:directive.page contentType = "text/html;charset = UTF-8"
9        pageEncoding = "UTF-8"/>
10    <f:view>
11       <ui:page binding = "#{ComponentesWeb.page1}" id = "page1">
12          <ui:html binding = "#{ComponentesWeb.html1}" id = "html1">
13             <ui:head binding = "#{ComponentesWeb.head1}" id = "head1">
14                <ui:link binding = "#{ComponentesWeb.link1}" id = "link1"
15                   url = "/resources/stylesheet.css"/>
16             </ui:head>
```

Figura 26.17 | Formulario de registro que demuestra el uso de los componentes JSF. (Parte 1 de 3).

```
<ui:body binding = "#{ComponentesWeb.body1}" id = "body1"
   style = "-rave-layout: grid">
   <ui:form binding = "#{ComponentesWeb.form1}" id = "form1">
      <ui:staticText binding = "#{ComponentesWeb.encabezado}"
         id = "encabezado" style = "font-size: 18px; left: 48px;
         top: 24px; position: absolute" text = "Este es un
         formulario de registro de ejemplo."/>
      <ui:staticText binding = "#{ComponentesWeb.instrucciones}"
         id = "instrucciones" style = "font-style: italic;
         left: 48px; top: 60px; position: absolute" text =
         "Por favor complete todos los campos y haga clic en Registro."/>
      <ui:image binding = "#{ComponentesWeb.imagenUsuario}" id =
         "imagenUsuario" style = "left: 48px; top: 96px;
         position: absolute" url = "/resources/usuario.JPG"/>
      <h:panelGrid binding = "#{ComponentesWeb.gridPanel1}"
         columns = "4" id = "gridPanel1" style = "height: 120px;
         left: 48px; top: 120px; position: absolute"
         width = "576">
         <ui:image binding = "#{ComponentesWeb.imagenNombre}"
            id = "imagenNombre" url = "/resources/nombre.JPG"/>
         <ui:textField binding = "#{ComponentesWeb.cuadroTextoNombre}"
            id = "cuadroTextoNombre"/>
         <ui:image binding = "#{ComponentesWeb.imagenApellido}"
            id = "imagenApellido" url = "resources/apellido.JPG"/>
         <ui:textField binding = "#{ComponentesWeb.cuadroTextoApellido}"
            id = "cuadroTextoApellido"/>
         <ui:image binding = "#{ComponentesWeb.imagenEmail}"
            id = "imagenEmail" url = "/resources/email.JPG"/>
         <ui:textField binding = "#{ComponentesWeb.cuadroTextoEmail}"
            id = "cuadroTextoEmail"/>
         <ui:image binding = "#{ComponentesWeb.imagenTelefono}"
            id = "imagenTelefono" url = "/resources/telefono.JPG"/>
         <ui:textField binding = "#{ComponentesWeb.cuadroTextoTelefono}"
            id = "cuadroTextoTelefono" label = "Debe tener la forma (555)
            555-5555"/>
      </h:panelGrid>
      <ui:image binding = "#{ComponentesWeb.imagenPublicaciones}"
         id = "imagenPublicaciones" style = "left: 48px; top: 264px;
         position: absolute" url =
         "/resources/publicaciones.JPG"/>
      <ui:staticText binding =
         "#{ComponentesWeb.etiquetaPublicacion}" id =
         "etiquetaPublicacion" style = "position: absolute;
         left: 300px; top: 264px" text = "De que libro desea obtener
         informacion?"/>
      <ui:dropDown binding = "#{ComponentesWeb.librosDesplegable}"
         id = "librosDesplegable" items = "#{ComponentesWeb.
         librosDesplegableDefaultOptions.options}" style = "left:
         48px; top: 300px; position: absolute”width:240px" />
      <ui:hyperlink binding = "#{ComponentesWeb.librosVinculo}"
         id = "librosVinculo" style = "left: 48px; top: 348px;
         position: absolute" target = "_blank" text = "Haga clic
         aqui para ver mas informacion acerca de nuestros libros."
         url = "http://www.deitel.com"/>
      <ui:image binding = "#{ComponentesWeb.image1}" id =
         "image1" style = "left" 48px; top: 396px;
         position: absolute" url = "/resources/so.JPG"/>
      <ui:staticText binding = "#{ComponentesWeb.etiquetaSO}" id =
```

Figura 26.17 | Formulario de registro que demuestra el uso de los componentes JSF. (Parte 2 de 3).

```
                        "etiquetaSO" style = "position: absolute; left: 252px;
                        top: 396px" text = "Que sistema operativo
                        utiliza?"/>
                     <ui:radioButtonGroup binding=
                        "#{ComponentesWeb.botonesSeleccionSO}" id =
                        "botonesSeleccionSO" items = "#{ComponentesWeb.
                        botonesSeleccionSODefaultOptions.options}" style =
                        "left: 48px; top: 432px; position: absolute"/>
                     <ui:button binding = "#{ComponentesWeb.botonRegistro}"
                        id = "botonRegistro" style = "left: 47px; top: 564px;
                        position: absolute" text = "Registro"/>
                  </ui:form>
               </ui:body>
            </ui:html>
         </ui:page>
      </f:view>
   </jsp:root>
```

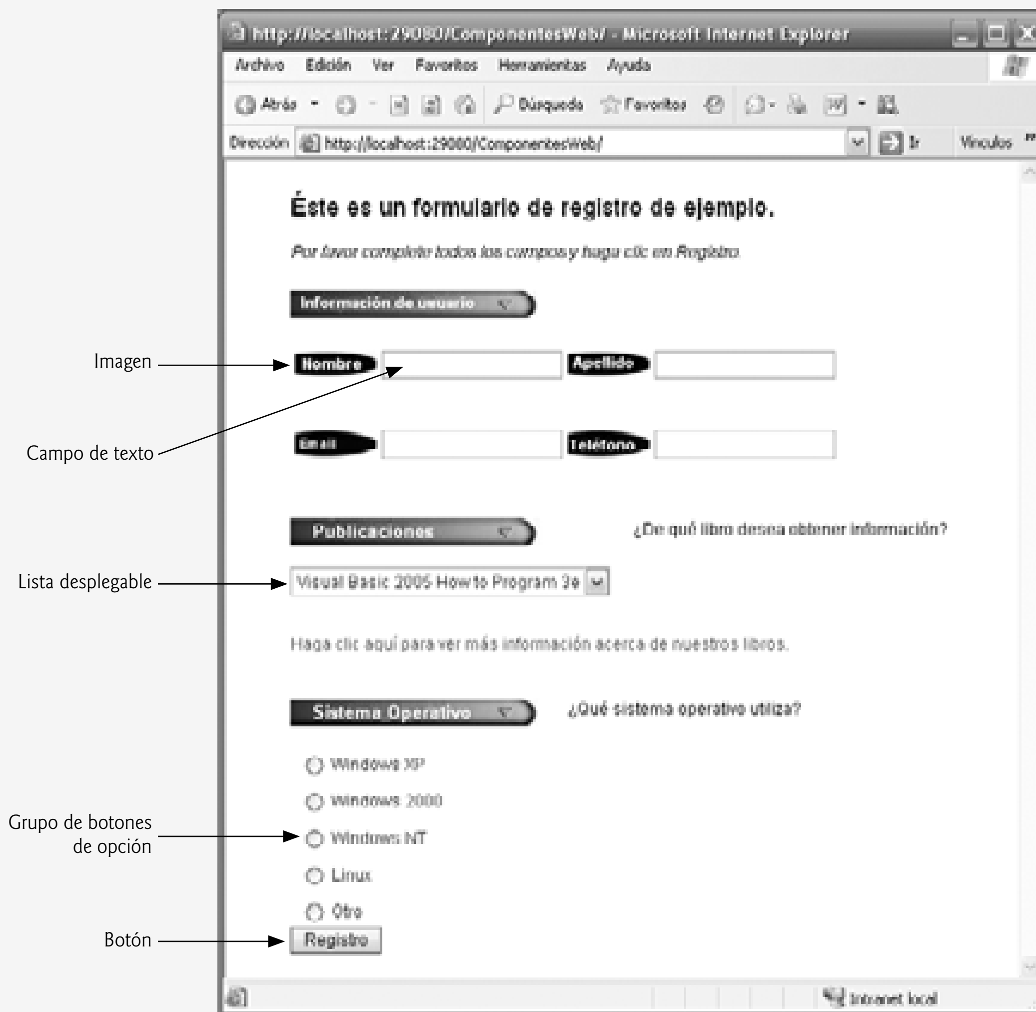


Figura 26.17 | Formulario de registro que demuestra el uso de los componentes JSF. (Parte 3 de 3).

Cómo agregar un componente de formato a una página Web

Para crear el esquema para la sección **Información del usuario** del formulario que se muestra en la figura 26.17, arrastre un componente **Panel de cuadrícula** en la página. En la ventana **Propiedades**, establezca la propiedad `columns` del componente en `4`. El componente también tiene propiedades para controlar el relleno de las celdas, el espaciado y otros elementos relacionados con la apariencia del componente. En este caso, acepte los valores predeterminados para estas propiedades. Ahora, simplemente puede arrastrar los componentes **Imagen** y **Campo de texto** para la información del usuario en el **Panel de cuadrícula**. Este componente administrará su espaciado y su organización en filas y columnas.

Cómo analizar los componentes Web en un formulario de registro de ejemplo

En las líneas 28 a 30 de la figura 26.17 se define un componente **Imagen**, un objeto de la clase `Image` que inserta una imagen en una página Web. Las imágenes que se utilizan en este ejemplo se encuentran en el directorio de ejemplos de este capítulo. Las imágenes que se van a mostrar en una página Web se deben colocar en la carpeta `resources` del proyecto. Para agregar imágenes al proyecto, suelte un componente **Imagen** en la página y haga clic en el botón de elipse que está a un lado de la propiedad **url** en la ventana **Propiedades**. A continuación se abrirá un cuadro de diálogo, en el que puede seleccionar la imagen a mostrar. Como aún no se han agregado imágenes a la carpeta `resources`, haga clic en el botón **Agregar archivo**, localice la imagen en el sistema de archivos de su computadora y haga clic en **Agregar archivo**. A continuación se copiará el archivo que usted seleccionó en el directorio `resources` del proyecto. Ahora puede seleccionar la imagen de la lista de archivos en la carpeta `resources` y hacer clic en **Aceptar** para insertar la imagen en la página.

Las líneas 31 a 52 contienen un elemento `h:panelGrid`, el cual representa al componente **Panel de cuadrícula**. Dentro de este elemento hay ocho componentes **Imagen** y **Campo de texto**. Los componentes **Campo de texto** nos permiten obtener la entrada de texto del usuario. Por ejemplo, en las líneas 37 y 38 se define un control **Campo de texto** que se utiliza para recolectar el nombre de pila del usuario. En las líneas 49 a 51 se define un **Campo de texto** con la propiedad `label` establecida en `"Debe tener la forma (555) 555-5555"`. Al establecer la propiedad `label` de un **Campo de texto**, se coloca el texto directamente encima de este componente. De manera alternativa, para etiquetar un **Campo de texto** puede arrastrar y soltar un componente **Etiqueta** en la página, lo cual le permitirá personalizar la posición y el estilo del componente **Etiqueta**.

El orden en el que se arrastran los componentes **Campo de texto** a la página es importante, ya que sus etiquetas JSP se agregan al archivo JSP en ese orden. Cuando un usuario oprime la tecla *Tab* para navegar de un campo de entrada a otro, navegarán por los campos en el orden en el que se hayan agregado las etiquetas JSP al archivo JSP. Para especificar el orden de navegación, debe arrastrar los componentes a la página en ese orden. De manera alternativa, puede establecer la propiedad **Tab Index** de cada campo de texto en la ventana **Propiedades**, para controlar el orden en el que el usuario avanzará mediante la tecla *Tab* por los campos de texto. Un componente con un índice de tabulación de 1 será el primero en la secuencia de tabulaciones.

En las líneas 62 a 65 se define una **Lista desplegable**. Cuando un usuario hace clic en la lista desplegable, expande y muestra una lista, en la que el usuario puede seleccionar un elemento. Este componente es un objeto de la clase `DropDownList` y está enlazado al objeto `librosDesplegable`, un objeto `SingleSelectOptionsList` que controla la lista de opciones. Este objeto se puede configurar de manera automática, haciendo clic con el botón derecho en la lista desplegable en modo **Diseño** y seleccionando **Configurar opciones predeterminadas**, con lo cual se abrirá el cuadro de diálogo **Personalizador de opciones** para agregar opciones a la lista. Cada opción consiste en un objeto `String` para mostrar, el cual representará la opción en el navegador, y de un objeto `String` de valor, el cual se devolverá cuando se obtenga la selección del usuario de la lista desplegable, por medio de programación. Java Studio Creator 2 construye el objeto `SingleSelectOptionsList` en el archivo de bean de página, con base en los pares mostrar-valor introducidos en el cuadro de diálogo **Personalizador de opciones**. Para ver el código que construye al objeto, cierre el cuadro de diálogo haciendo clic en **Aceptar**, abra el archivo de bean de página y expanda el nodo **Creator-managed Component Definition** cerca de la parte superior del archivo. El objeto se construye en el método `_init`, el cual se llama desde el método `init` la primera vez que se carga la página.

El componente **Hipervínculo** (líneas 66 a 70) de la clase `Hyperlink` agrega un vínculo a una página Web. La propiedad `url` de este componente especifica el recurso (`http://www.deitel.com` en este caso) que se solicita cuando un usuario hace clic en el hipervínculo. Al establecer la propiedad `target` en `_blank`, especificamos que la página Web solicitada debe abrirse en una nueva ventana del navegador. De manera predeterminada, los componentes **Hipervínculo** hacen que las páginas se abran en la misma ventana del navegador.

En las líneas 78 a 82 se define un componente **Grupo de botones de selección** de la clase `RadioButtonGroup`, el cual proporciona una serie de botones de opción, de los cuales el usuario sólo puede seleccionar uno. Al igual que **Lista desplegable**, un **Grupo de botones de selección** está enlazado a un objeto `SingleSelectOptionsList`. Para editar las opciones, haga clic con el botón derecho del ratón en el componente y seleccione **Configurar opciones predeterminadas**. Al igual que la lista desplegable, el IDE genera el constructor del objeto `SingleSelectOptionsList` automáticamente y lo coloca en el método `_init` de la clase de bean de página.

El último control Web en la figura 26.17 es un **Botón** (líneas 83 a 85), un componente JSF de la clase `Button` que desencadena una acción cuando es oprimido. Por lo general, un componente **Botón** se asigna a un elemento **input** de XHTML, en donde su atributo `type` se establece en `submit`. Como dijimos antes, al hacer clic en el botón **Registro** en este ejemplo no se produce ninguna acción.

26.6.2 Validación mediante los componentes de validación y los validadores personalizados

En esta sección presentamos la **validación** de formularios. La validación de la entrada del usuario es un importante paso para recolectar la información de los usuarios. La validación ayuda a evitar los errores de procesamiento debido a que los datos de entada del usuario estén incompletos, o tengan un formato inapropiado. Por ejemplo, puede realizar la validación para asegurar que se hayan completado todos los campos requeridos, o que un campo de código postal contenga exactamente cinco dígitos. Java Studio Creator 2 proporciona tres componentes de validación. Un **Validador de longitud** determina si un campo contiene un número aceptable de caracteres. El **Validador de intervalo doble** y el **Validador de intervalo largo** determinan si la entrada numérica se encuentra dentro de intervalos aceptables. El paquete **javax.faces.validators** contiene las clases para estos validadores. Studio Creator 2 también permite la validación personalizada con métodos de validación en el archivo de bean de página. El siguiente ejemplo demuestra la validación mediante el uso de un componente de validación y de la validación personalizada.

Cómo validar los datos de un formulario en una aplicación Web

El ejemplo en esta sección pide al usuario que introduzca su nombre, dirección de e-mail y número telefónico. Después de que el usuario introduce los datos, pero antes de que éstos se envíen al servidor Web, la validación nos asegura que el usuario haya introducido un valor en cada campo, que el nombre introducido no exceda a 30 caracteres y que la dirección de e-mail y el número telefónico se encuentren en un formato aceptable. En este ejemplo, (555) 123-4567, 555-123-4567 y 123-4567 se consideran números telefónicos válidos. Una vez que se envían los datos, el servidor Web responde mostrando un mensaje apropiado y un componente **Panel de cuadrícula** que repite la información enviada. Observe que una aplicación comercial real, por lo general, almacena los datos enviados en una base de datos o en el servidor. Nosotros simplemente enviamos de vuelta los datos a la página, para demostrar que el servidor recibió los datos.

Creación de la página Web

Esta aplicación web introduce dos componentes JSF adicionales: **Etiqueta** y **Mensaje**, de la sección de componentes **Básicos** de la **Paleta**. Cada uno de los tres campos de texto debe tener su propia etiqueta y su propio mensaje. Los componentes **Etiqueta** describen a otros componentes, y se pueden asociar con los campos de entrada del usuario si se establece su propiedad `for`. Los componentes **Mensaje** muestran mensajes de error cuando falla la validación. Esta página requiere tres componentes **Campo de texto**, tres componentes **Etiqueta** y tres componentes **Mensaje**, así como un componente **Botón** para enviar los datos. Para asociar los componentes **Etiqueta** y **Mensaje** con sus correspondientes componentes **Campo de texto**, mantenga oprimidas las teclas *Ctrl* y *Mayús*, y después arrastre la etiqueta o mensaje hacia el **Campo de texto** apropiado. En la ventana **Propiedades**, observe que la propiedad `for` de cada **Etiqueta** y **Mensaje** se encuentra establecida con el componente **Campo de texto** apropiado.

También es conveniente agregar un componente **Texto estático** para mostrar un mensaje de éxito en la validación al final de la página. Establezca el texto en `"Gracias por enviar sus datos. <br/> Recibimos la siguiente información:"` y cambie el `id` del componente a `textoResultado`. En la ventana **Propiedades**, desactive las propiedades `rendered` y `escape` del componente. La propiedad **rendered** controla si el componente se mostrará la primera vez que se cargue la página. Al establecer **escaped** en `false`, el navegador podrá reconocer la etiqueta `<br/>`, de manera que pueda empezar una nueva línea de texto, en vez de mostrar los caracteres `"<br/>"` en la página web.

Por último, agregue un componente **Panel de cuadrícula** debajo del componente `textoResultado`. El panel debe tener dos columnas, una para mostrar componentes **Texto estático** que etiqueten los datos validados del usuario, y uno para mostrar componentes **Texto estático** que vuelvan a imprimir esos datos.

El archivo JSP para esta página se muestra en la figura 26.18. En las líneas 30 a 34, 35 a 39 y 40 a 44 se definen elementos `ui:textField` para obtener el nombre del usuario, la dirección de e-mail y el número telefónico, respectivamente. En las líneas 45 a 48, 49 a 53 y 54 a 58 se definen elementos `ui:label` para cada uno de estos campos de texto. En las líneas 63 a 74 se definen los elementos `ui:message` de los campos de texto. En las líneas 59 a 62 se define un elemento `ui:button` llamado **Enviar**. En las líneas 75 a 80 se crea un elemento `ui:staticText` llamado `textoResultado`, el cual muestra la respuesta del servidor cuando el usuario envía el formulario con éxito, y en las líneas 81 a 101 se define un elemento `ui:panel de cuadricula` que contiene componentes para repetir la entrada validada del usuario y mostrarla de nuevo en el navegador.

```
1  <?xml version = "1.0" encoding = "UTF-8"?>
2
3  <!-- Fig. 26.18: Validacion.jsp -->
4  <!-- JSP que demuestra la validación de la entrada del usuario. -->
5  <jsp:root version = "1.2" xmlns:f = "http://java.sun.com/jsf/core"
6     xmlns:h = "http://java.sun.com/jsf/html" xmlns:jsp =
7     "http://java.sun.com/JSP/Page" xmlns:ui = "http://www.sun.com/web/ui">
8     <jsp:directive.page contentType = "text/html; charset = UTF-8"
9        pageEncoding = "UTF-8"/>
10    <f:view>
11       <ui:page binding = "#{Validacion.page1}" id = "page1">
12          <ui:html binding = "#{Validacion.html1}" id = "html1">
13             <ui:head binding = "#{Validacion.head1}" id = "head1"
14                title = "Validación">
15                <ui:link binding = "#{Validacion.link1}" id = "link1"
16                   url = "/resources/stylesheet.css"/>
17             </ui:head>
18             <ui:body binding = "#{Validacion.body1}" focus = "form1.ctextoNombre"
19                id = "body1" style = "-rave-layout: grid">
20                <ui:form binding = "#{Validacion.form1}" id = "form1">
21                   <ui:staticText binding = "#{Validacion.encabezado}" id =
22                      "encabezado" style = "font-size: 18px; height: 22px;
23                      left: 24px; top: 24px; position: absolute; widht:456px"
24                      text = "Por favor complete el siguiente formulario."/>
25                   <ui:staticText binding = "#{Validacion.instrucciones}"
26                      id = "instrucciones" style = "font-size: 14px;
27                      font-style: italic; left: 24px; top: 60px; position:
28                      absolute; width: 406 px" text = "Todos los campos son requeridos
29                      y deben contener información válida."/>
30                   <ui:textField binding = "#{Validacion.ctextoNombre}" columns =
31                      "30" id = "ctextoNombre" required = "true" style = "left:
32                      180px; top: 96px; position: absolute; width: 216px"
33                      validator =
34                      "#{Validacion.longitudNombreValidador.validate}"/>
35                   <ui:textField binding = "#{Validacion.ctextoEmail}"
36                      columns = "28" id = "ctextoEmail" required = "true"
37                      style = "left: 180px; top: 144px; position: absolute;
38                      width: 216px" validator =
39                      "#{Validacion.ctextoEmail_validate}"/>
40                   <ui:textField binding = "#{Validacion.ctextoTelefono}"
41                      columns = "30" id = "ctextoTelefono" required = "true"
42                      style = "left: 180px; top: 192px; position: absolute;
43                      width: 216px" validator =
44                      "#{Validacion.ctextoTelefono_validate}"/>
```

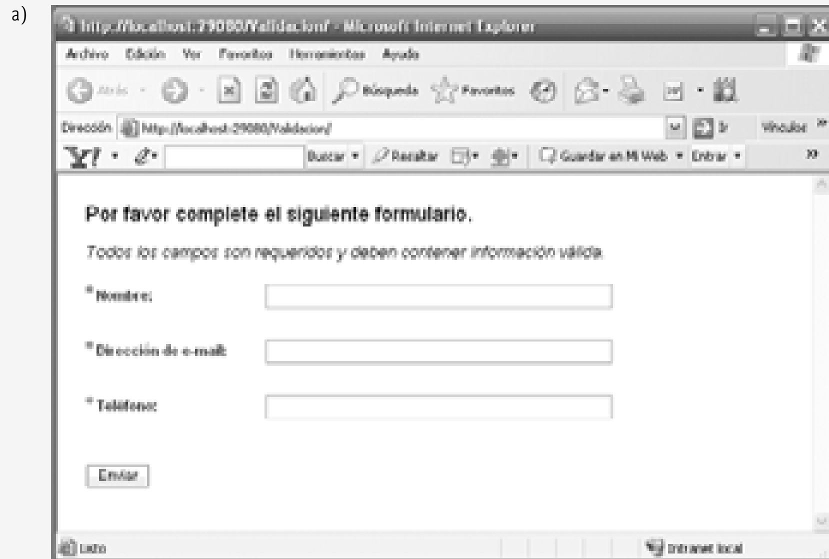
Figura 26.18 | JSP que demuestra la validación de la entrada del usuario. (Parte 1 de 4).

```
                  <ui:label binding = "#{Validacion.etiquetaNombre}" for =
                     "ctextoNombre" id = "etiquetaNombre" style = "font-weight:
                     normal; height: 24px; left: 24px; top: 96px;
                     position: absolute; width: 94px" text = "Nombre:"/>
                  <ui:label binding = "#{Validacion.etiquetaEmail}" for =
                     "ctextoEmail" id = "etiquetaEmail" style = "font-weight:
                     normal; height: 24px; left: 24px; top: 144px;
                     position: absolute; width: 142px" text =
                     "Dirección de e-mail: "/>
                  <ui:label binding = "#{Validacion.etiquetaTelefono}" for=
                     "ctextoTelefono" id = "etiquetaTelefono" style = "font-weight:
                     normal; height: 24px; left: 24px; top: 192px
                     position: absolute; width: 142px" text =
                     "Teléfono:"/>
                  <ui:button action = "#{Validacion.botonEnviar_action}"
                     binding = "#{Validacion.botonEnviar}" id =
                     "botonEnviar" style = "position: absolute; left: 24px;
                     top: 240px" text = "Enviar"/>
                  <ui:message binding = "#{Validacion.mensajeEmail}" for =
                     "ctextoEmail" id = "mensajeEmail" showDetail = "false"
                     showSummary = "true" style = "left: 504px; top:
                     144px; position: absolute"/>
                  <ui:message binding = "#{Validacion.mensajeTelefono}" for =
                     "ctextoTelefono" id = "mensajeTelefono" showDetail= "false"
                     showSummary = "true" style = "left: 504px; top:
                     192px; position: absolute"/>
                  <ui:message binding = "#{Validacion.mensajeNombre}" for =
                     "ctextoNombre" id = "mensajeNombre" showDetail = "false"
                     showSummary = "true" style = "left: 504px; top: 96px;
                     position: absolute"/>
                  <ui:staticText binding = "#{Validacion.textoResultado}"
                     escape = "false" id = "textoResultado" rendered = "false"
                     style = "height: 46px; left: 24px; top: 312px;
                     position: absolute; width: 312px" text = "Gracias por
                     enviar sus datos. &lt;br/&gt;Recibimos
                     la siguiente información:"/>
                  <h:panelGrid binding = "#{Validacion.panelCuadricula}"
                     columns = "2" id = "panelCuadricula" rendered = "false"
                     style = "background-color: seashell; height: 120px;
                     left: 24px; top: 360px; position: absolute"
                     width = "360">
                     <ui:staticText binding =
                        "#{Validacion.etiquetaResultadoNombre}" id=
                        "etiquetaResultadoNombre" text= "Nombre:"/>
                     <ui:staticText binding = "#{Validacion.resultadoNombre}"
                        id = "resultadoNombre"/>
                     <ui:staticText binding =
                        "#{Validacion.etiquetaResultadoEmail}" id =
                        "etiquetaResultadoEmail" text = "E-mail: "/>
                     <ui:staticText binding = "#{Validacion.resultadoEmail}"
                        id= "resultadoEmail"/>
                     <ui:staticText binding =
                        "#{Validacion.etiquetaResultadoTelefono}" id =
                        "etiquetaResultadoTelefono" text = "Teléfono: "/>
                     <ui:staticText binding = "#{Validacion.resultadoTelefono}"
                        id = "resultadoTelefono"/>
                  </h:panelGrid>
               </ui:form>
            </ui:body>
```

Figura 26.18 | JSP que demuestra la validación de la entrada del usuario. (Parte 2 de 4).

```
            </ui:html>
        </ui:page>
    </f:view>
</jsp:root>
```

a)



b)



Figura 26.18 | JSP que demuestra la validación de la entrada del usuario. (Parte 3 de 4).

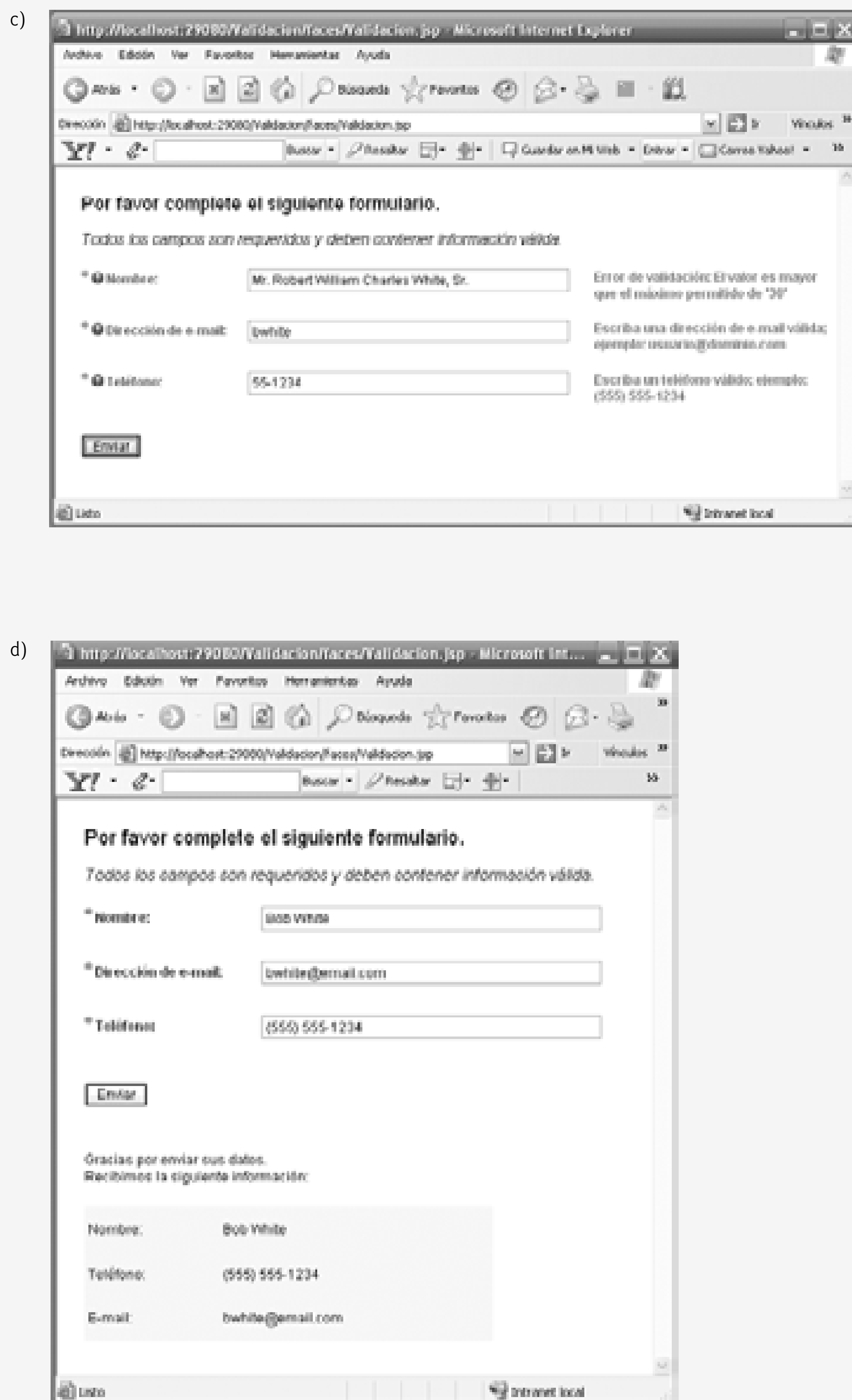


Figura 26.18 | JSP que demuestra la validación de la entrada del usuario. (Parte 4 de 4).

Cómo establecer la propiedad `Required` de un componente de entrada

Asegurarse que el usuario haya realizado una selección, o que haya escrito texto en un elemento de entrada requerido, es un tipo básico de validación. Para ello, hay que activar la casilla `required` en la ventana **Properties** del elemento. Si agrega un componente de validación o un método de validación personalizado a un campo de entrada, la propiedad `required` del campo debe establecerse en `true` para que se lleve a cabo la validación. Observe que los tres elementos `ui:textField` de entrada en este ejemplo (figura 26.18, líneas 30 a 44) tienen su propiedad `required` establecida en `true`. Observe además en el Editor visual que la etiqueta para un campo requerido se marca automáticamente mediante un asterisco color rojo. Si un usuario envía este formulario con campos de texto vacíos, se mostrará el mensaje de error predeterminado para un campo requerido en el componente `ui:message` asociado del campo vacío.

Uso del componente `LengthValidator`

En este ejemplo, utilizamos el componente **Validador de longitud** (el cual se encuentra en la sección **Validadores** de la **Paleta**) para asegurar que la longitud del nombre del usuario no exceda a 30 caracteres. Esto podría ser útil para asegurar que un valor pueda guardarse en un campo específico de la base de datos.

Para agregar un **Validador de longitud** a un componente, simplemente arrastre el validador de la **Paleta** y suéltelo en el campo a validar. A continuación, aparecerá un nodo **lengthValidator** en la sección **Validación** de la ventana **Esquema**. Para editar las propiedades del componente de validación, haga clic en este nodo y establezca las propiedades `maximum` y `minimum` en el número deseado de caracteres en la ventana **Propiedades**. Aquí sólo estableceremos la propiedad `maximum` en 30. También modificamos el `id` del componente a `longitudNombreValidador`. Observe que el campo de entrada `ctextoNombre` en el archivo JSP se ha enlazado al método `validate` de la propiedad `longitudNombreValidador` en el archivo de bean de página (líneas 33 y 34).

Este validador permite a los usuarios escribir todo el texto que deseen en el campo y, si exceden el límite, se mostrará el mensaje de error de validación de longitud predeterminado en el componente `ui:message` del campo, después de que el usuario haga clic en el botón **Enviar**. Es posible limitar la longitud de la entrada del usuario sin validación. Al establecer la propiedad `maxLength` de un **Campo de texto**, el cursor de este componente no avanzará más allá del máximo número permisible de caracteres, por lo que el usuario no podrá enviar datos que excedan al límite de longitud.

Uso de expresiones regulares para realizar la validación personalizada

Algunas de las tareas de validación más comunes incluyen comprobar la entrada del usuario para el formato apropiado. Por ejemplo, tal vez sea necesario comprobar las direcciones de e-mail y los números telefónicos que se hayan introducido, para asegurar que se conformen al formato estándar para direcciones de e-mail y números telefónicos válidos. Comparar la entrada del usuario con una expresión regular es un método efectivo para asegurar que la entrada tenga un formato apropiado (en la sección 30.7 hablaremos sobre las expresiones regulares). Java Studio Creator 2 no proporciona componentes para validar mediante el uso de expresiones regulares, por lo que nosotros agregaremos nuestros propios métodos de validación al archivo de bean de página. Para agregar un validador personalizado a un componente de entrada, haga clic con el botón derecho del ratón sobre el componente y seleccione **Editar manejador de eventos > validate**. Esto crea un método de validación para el componente, con un cuerpo vacío en el archivo de bean de página. En breve agregaremos código a este método. Observe que los atributos `validate` de `ctextoEmail` y `ctextoTelefono` están enlazados con sus respectivos métodos de validación en el archivo de bean de página (líneas 38 a 39 y 43 a 44).

Análisis del archivo de bean de página para un formulario que reciba la entrada del usuario

La figura 26.19 contiene el archivo de bean de página para el archivo JSP de la figura 26.18. En la línea 33 se establece la longitud máxima para `longitudNombreValidator`, que es una propiedad de este bean de página. Recuerde que el campo de texto del nombre se enlazó a esta propiedad en el archivo JSP. Los métodos `ctextoEmail_validate` (líneas 398 a 410) y `ctextoTelefono_validate` (líneas 414 a 426) son los métodos validadores personalizados que verifican que el usuario haya introducido la dirección de e-mail y el número telefónico, respectivamente. El método `botonEnviar_action` (líneas 429 a 440) vuelve a imprimir de vuelta al usuario los datos introducidos, si la validación fue exitosa. Los métodos de validación se llaman antes del manejador de eventos, por lo que si la validación falla, no se hará la llamada a `botonEnviar_action` y la entrada del usuario no se repetirá.

Los dos métodos de validación en este archivo de bean de página validan el contenido de un campo de texto, comparándolo con una expresión regular mediante el método `match` de `String`, el cual recibe una expresión regular como argumento y devuelve `true` si ese objeto `String` se conforma con el formato especificado.

```
1   // Fig. 26.19: Validacion.java
2   // Bean de página para validar la entrada del usuario y volver a mostrar esa
3   // entrada si es válida.
4   package validacion;
5
6   import com.sun.rave.web.ui.appbase.AbstractPageBean;
7   import com.sun.rave.web.ui.component.Body;
8   import com.sun.rave.web.ui.component.Form;
9   import com.sun.rave.web.ui.component.Head;
10  import com.sun.rave.web.ui.component.Html;
11  import com.sun.rave.web.ui.component.Link;
12  import com.sun.rave.web.ui.component.Page;
13  import javax.faces.FacesException;
14  import com.sun.rave.web.ui.component.StaticText;
15  import com.sun.rave.web.ui.component.TextField;
16  import com.sun.rave.web.ui.component.TextArea;
17  import com.sun.rave.web.ui.component.Label;
18  import com.sun.rave.web.ui.component.Button;
19  import com.sun.rave.web.ui.component.Message;
20  import javax.faces.component.UIComponent;
21  import javax.faces.context.FacesContext;
22  import javax.faces.validator.ValidatorException;
23  import javax.faces.application.FacesMessage;
24  import javax.faces.component.html.HtmlPanelGrid;
25  import javax.faces.validator.LengthValidator;
26
27  public class Validacion extends AbstractPageBean
28  {
29     private int __placeholder;
30
31     private void _init() throws Exception
32     {
33        longitudNombreValidador.setMaximum( 30 );
34     } // fin del método _init
35
36     // Para ahorrar espacio, omitimos el código de las líneas 36 a 345. El código
37     // fuente completo se proporciona con los ejemplos de este capítulo.
38
346    public Validacion()
347    {
348       // constructor vacío
349    } // fin del constructor
350
351    protected ApplicationBean1 getApplicationBean1()
352    {
353       return (ApplicationBean1)getBean("ApplicationBean1");
354    } // fin del método getApplicationBean1
355
356    protected RequestBean1 getRequestBean1()
357    {
358       return (RequestBean1)getBean("RequestBean1");
359    } // fin del método getRequestBean1
```

Figura 26.19 | Bean de página para validar la entrada del usuario y volver a mostrar esa entrada, si es válida. (Parte 1 de 3).

```

   protected SessionBean1 getSessionBean1()
   {
      return (SessionBean1)getBean("SessionBean1");
   } // fin del método getSessionBean1

   public void init()
   {
      super.init();
      try
      {
         _init();
      } // fin de try
      catch (Exception e)
      {
         log(“Error de inicializacion de Validacion”, e);
         throw e instanceof FacesException ? (FacesException) e:
            new FacesException(e);
      } // fin de catch
   } // fin del método init

   public void preprocess()
   {
      // cuerpo vacío
   } // fin del método preprocess

   public void prerender()
   {
      // cuerpo vacío
   } // fin del método prerender

   public void destroy()
   {
      // cuerpo vacío
   } // fin del método destroy

   // valida la dirección de email introducida, comparándola con la expresión
   // regular que representa la forma de una dirección de email válida.
   public void ctextoEmail_validate(FacesContext context,
      UIComponent component, Object value)
   {
      String email = String.valueOf( value );

      // si la dirección de e-mail introducida no está en un formato válido
      if ( !email.matches(
         "\\w+([-+.']\\w+)*@\\w+([-.]\\w+)*\\.\\w+([-.]\\w+)*" ) )
      {
         throw new ValidatorException( new FacesMessage(
            "Escriba una direccion de e-mail valida; ejemplo: usuario@dominio.com" ) );
      } // fin de if
   } // fin del método ctextoEmail_validate

   // valida el número telefónico introducido, comparándolo con la expresión
   // regular que representa la forma de un número telefónico válido.
   public void ctextoTelefono_validate(FacesContext context,
      UIComponent component, Object value)
   {
      String telefono = String.valueOf( value );
```

Figura 26.19 | Bean de página para validar la entrada del usuario y volver a mostrar esa entrada, si es válida. (Parte 2 de 3).

```
418
419          // si el número telefónico introducido no está en un formato válido
420          if ( !telefono.matches(
421             "((\\(\\d{3}\\) ?)|(\\d{3}-))?\\d{3}-\\d{4}" ) )
422          {
423             throw new ValidatorException( new FacesMessage(
424                "Escriba un telefono valido; ejemplo: (555) 555-1234" ) );
425          } // fin de if
426       } // fin del método ctextoTelefono_validate
427
428       // muestra las entradas del formulario validadas en un Panel de cuadrícula
429       public String botonEnviar_action()
430       {
431          String nombre = String.valueOf( ctextoNombre.getValue() );
432          String email = String.valueOf( ctextoEmail.getValue() );
433          String telefono = String.valueOf( ctextoTelefono.getValue() );
434          resultadoNombre.setValue( nombre );
435          resultadoEmail.setValue( email );
436          resultadoTelefono.setValue( telefono );
437          panelCuadricula.setRendered( true );
438          textoResultado.setRendered( true );
439          return null;
440       } // fin del método botonEnviar_accion
441    } // fin de la clase Validacion
```

Figura 26.19 | Bean de página para validar la entrada del usuario y volver a mostrar esa entrada, si es válida. (Parte 3 de 3).

Para el método `ctextoEmail_validate`, usamos la siguiente expresión de validación:

```
\w+([-+.']\\w+)*@\\w+([-.]\\w+)*\\.\\w+([-.]\\w+)*
```

Observe que cada barra diagonal inversa en la expresión regular `String` (línea 405) se debe escapar con otra barra diagonal inversa (como en \\), ya que el carácter de barra diagonal inversa normalmente representa el inicio de una secuencia de escape. Esta expresión regular indica que una dirección de e-mail es válida si la parte antes del símbolo `@` contiene uno o más caracteres de palabra (es decir, caracteres alfanuméricos o de guión bajo), seguidos de uno o más objetos `String` compuestos de un guión corto, signo más, punto o apóstrofo (') y de más caracteres de palabra. Después del símbolo `@`, una dirección de e-mail válida debe contener uno o más grupos de caracteres de palabras, que pueden estar separados por guiones cortos o puntos, seguidos de un punto requerido y de otro grupo de uno o más caracteres, que pueden estar separados por guiones cortos o puntos. Por ejemplo, las direcciones de e-mail `bob's-personal.email@white.email.com`, `bob-white@my-email.com` y `bob.white@email.com` son todas válidas. Si el usuario escribe texto en `ctextoEmail` que no tenga el formato correcto y trata de enviar el formulario, en las líneas 407 y 408 se lanza una excepción `ValidatorException`. El componente `mensajeEmail` atrapará esta excepción y mostrará el mensaje en color rojo.

La expresión regular en `ctextoTelefono_validate` asegura que el componente `ctextoTelefono` contenga un número telefónico válido antes de enviar el formulario. La entrada del usuario se compara con la expresión regular

```
((\\(\\d{3}\\) ?)|(\\d{3}-))?\\d{3}-\\d{4}
```

(De nuevo, cada barra diagonal inversa se escapa en la expresión regular `String` de la línea 421). Esta expresión indica que un número telefónico puede contener un código de área de tres dígitos, ya sea entre paréntesis o no, y debe ir seguido de un espacio opcional, o sin paréntesis y seguido por un guión corto obligatorio. Después de un código de área opcional, un número telefónico debe contener tres dígitos, un guión corto y otros cuatro dígitos. Por ejemplo, `(555) 123-4567`, `555-123-4567` y `123-4567` son todos números telefónicos válidos. Si un usuario escribe un número telefónico inválido, en las líneas 423 y 424 se lanza una excepción `ValidatorException`. El componente `mensajeTelefono` atrapa esta excepción y muestra el mensaje de error en color rojo.

Si todos los seis validadores tienen éxito (es decir, que cada componente `TextField` contenga datos, que el nombre tenga menos de 30 caracteres y que la dirección de e-mail y el número telefónico sean válidos), al hacer clic en el botón **Enviar** se enviarán los datos del formulario al servidor. Como se muestra en la figura 26.18(d), el método `botonEnviar_action` muestra los datos enviados en un `panelCuadricula` (líneas 434 a 437) y un mensaje de éxito en `textoResultado` (línea 438).

26.7 Rastreo de sesiones

En los primeros días de Internet, los comercios electrónicos no podían proveer el tipo de servicio personalizado que comúnmente se experimenta en las tiendas reales. Para lidiar con este problema, los comercios electrónicos empezaron a establecer mecanismos mediante los cuales pudieran personalizar las experiencias de navegación de los usuarios, preparando contenido a la medida de los usuarios individuales, permitiéndoles ignorar al mismo tiempo la información irrelevante. Para lograr este nivel de servicio, los comercios rastrean el movimiento de cada cliente a través de sus sitios Web y combinan los datos recolectados con la información que proporciona el consumidor, incluyendo la información de facturación y las preferencias personales, intereses y pasatiempos.

Personalización

La **personalización** hace posible que los comercios electrónicos se comuniquen con eficiencia con sus clientes, y también mejora la habilidad del usuario para localizar los productos y servicios deseados. Las compañías que proporcionan contenido de interés especial para los usuarios pueden establecer relaciones con los clientes, y fomentar esas relaciones con el paso del tiempo. Además, al enviar a los clientes ofertas personales, recomendaciones, anuncios, promociones y servicios, los comercios electrónicos crean una lealtad en los clientes. Los sitios Web pueden utilizar tecnología sofisticada para permitir a los visitantes personalizar las páginas de inicio para satisfacer sus necesidades y preferencias personales. De manera similar, los sitios de compras en línea comúnmente almacenan la información personal para los clientes, notificaciones personalizadas y ofertas especiales de acuerdo con sus intereses. Dichos servicios alientan a los clientes a visitar los sitios y realizar compras con más frecuencia.

Privacidad

Sin embargo, existe una concesión entre el servicio de comercio electrónico personalizado y la protección de la privacidad. Algunos consumidores adoptan la idea del contenido personalizado, pero otros temen a las posibles consecuencias adversas, si la información que proporcionan a los comercios electrónicos es liberada o recolectada por tecnologías de rastreo. Los consumidores y los defensores de la privacidad preguntan: ¿Qué pasa si el comercio electrónico al que proporcionamos nuestros datos personales vende o proporciona esa información a otra organización, sin nuestro consentimiento? ¿Qué pasa si no queremos que nuestras acciones en Internet (un medio supuestamente anónimo) sean rastreadas y registradas por terceros desconocidos? ¿Qué pasa si personas no autorizadas obtienen acceso a los datos privados delicados, como los números de tarjetas de crédito o el historial médico? Todas estas son preguntas con las que los programadores, consumidores, comercios electrónicos y legisladores deben debatir y lidiar.

Cómo reconocer a los clientes

Para proporcionar servicios personalizados a los consumidores, los comercios electrónicos deben tener la capacidad de reconocer a los clientes cuando solicitan información de un sitio. Como hemos visto antes, el sistema de petición/respuesta en el que opera la Web se lleva a cabo mediante HTTP. Por desgracia, HTTP es un protocolo sin estado; no soporta conexiones persistentes que permitan a los servidores Web mantener información de estado, en relación con clientes específicos. Por lo tanto, los servidores Web no pueden determinar si una petición proviene de un cliente específico, o si una serie de peticiones provienen de uno o varios clientes. Para sortear este problema, los sitios pueden proporcionar mecanismos para identificar a los clientes individuales. Una sesión representa a un cliente único en un sitio Web. Si el cliente sale de un sitio y regresa después, aún será reconocido como el mismo usuario. Para ayudar al servidor a diferenciar un cliente de otro, cada cliente debe identificarse a sí mismo con el servidor. El rastreo de clientes individuales, conocido como **rastreo de sesiones**, puede lograrse de varias formas. Una técnica popular utiliza cookies (sección 26.7.1); otra utiliza el objeto `SessionBean` (sección 26.7.2). Otras técnicas adicionales de rastreo de sesiones incluyen el uso de elementos `input form` de tipo `"hidden"` y la reescritura de URLs. Con los elementos `"hidden"`, un formulario Web puede escribir los datos de rastreo de sesión en un componente `form` en la página Web que devuelve al cliente, en respuesta a una petición

previa. Cuando el usuario envía el formulario en la nueva página Web, todos los datos del formulario (incluyendo los campos `"hidden"`) se envían al manejador del formulario en el servidor Web. Con la reescritura de URLs, el servidor Web incrusta la información de rastreo de sesión directamente en los URLs de los hipervínculos en los que el usuario hace clic para enviar las subsiguientes peticiones al servidor Web.

26.7.1 Cookies

Las **cookies** proporcionan a los desarrolladores Web una herramienta para personalizar las páginas Web. Una cookie es una pieza de datos que, por lo general, se almacena en un archivo de texto en la computadora del usuario. Una cookie mantiene información acerca del cliente, durante y entre las sesiones del navegador. La primera vez que un usuario visita el sitio Web, su computadora podría recibir una cookie; después, esta cookie se reactiva cada vez que el usuario vuelve a visitar ese sitio. La información recolectada tiene el propósito de ser un registro anónimo que contiene datos, los cuales se utilizan para personalizar las visitas futuras del usuario al sitio Web. Por ejemplo, las cookies en una aplicación de compras podría almacenar identificadores únicos para los usuarios. Cuando un usuario agregue elementos a un carrito de compras en línea, o cuando realice alguna otra tarea que origine una petición al servidor Web, éste recibe una cookie del cliente, la cual contiene el identificador único del usuario. Después, el servidor utiliza el identificador único para localizar el carrito de compras y realizar cualquier procesamiento requerido.

Además de identificar a los usuarios, las cookies también pueden indicar las preferencias de compra del usuario. Cuando un servidor Web recibe una petición de un cliente, el servidor puede analizar la(s) cookie(s) que envió al cliente durante las sesiones previas de comunicación, con lo cual puede identificar las preferencias del cliente y mostrar de inmediato productos que sean de su interés.

Cada interacción basada en HTTP entre un cliente y un servidor incluye un encabezado, el cual contiene información sobre la petición (cuando la comunicación es del cliente al servidor) o sobre la respuesta (cuando la comunicación es del servidor al cliente). Cuando una página recibe una petición, el encabezado incluye información como el tipo de petición (por ejemplo, `GET` o `POST`) y cualquier cookie que se haya enviado anteriormente del servidor, para almacenarse en el equipo cliente. Cuando el servidor formula su respuesta, la información del encabezado contiene cualquier cookie que el servidor desee almacenar en la computadora cliente, junto con más información, como el tipo MIME de la respuesta.

La **fecha de expiración** de una cookie determina la forma en que ésta permanecerá en la computadora del cliente. Si no establecemos una fecha de expiración para la cookie, el navegador Web mantendrá la cookie mientras dure la sesión de navegación. En caso contrario, el navegador Web mantendrá la cookie hasta que llegue la fecha de expiración. Cuando el navegador solicita un recurso de un servidor Web, las cookies que el servidor Web envió previamente al cliente se devuelven al servidor como parte de la petición formulada por el navegador. Las cookies se eliminan cuando **expiran**.



Tip de portabilidad 26.1

Los clientes pueden deshabilitar las cookies en sus navegadores Web para tener más privacidad. Cuando esos clientes utilicen aplicaciones Web que dependan de las cookies para mantener la información de estado, las aplicaciones no se ejecutarán correctamente.

Uso de cookies para proporcionar recomendaciones de libros

La siguiente aplicación Web muestra cómo utilizar cookies. El ejemplo contiene dos páginas. En la primera página (figuras 26.20 y 26.22), los usuarios seleccionan un lenguaje de programación favorito de un grupo de botones de opción y envían el formulario al servidor Web, para que éste lo procese. El servidor Web responde creando una cookie que almacena el lenguaje seleccionado y el número ISBN para un libro recomendado sobre ese tema. Después, el servidor despliega nuevos componentes en el navegador, que permiten al usuario seleccionar otro lenguaje de programación favorito o ver la segunda página en nuestra aplicación (figuras 26.23 y 26.24), la cual enlista los libros recomendados que pertenezcan al (los) lenguaje(s) de programación que el usuario haya seleccionado. Cuando el usuario hace clic en el hipervínculo, las cookies previamente almacenadas en el cliente se leen y se utilizan para formar la lista de recomendaciones de libros.

El archivo JSP de la figura 26.20 contiene un **Grupo de botones de selección** (líneas 26 a 39) con las opciones **Java, C, C++, Visual Basic 2005** y **Visual C# 2005**. Recuerde que puede establecer los objetos `String` Mostrar

y Valor de los botones de opción haciendo clic derecho en **Grupo de botones de selección** y seleccionando **Configurar opciones predeterminadas.** Para seleccionar un lenguaje de programación, el usuario debe hacer clic en uno de los botones de opción. Cuando el usuario oprime el botón **Enviar**, la aplicación Web crea una cookie que contiene el lenguaje seleccionado. Esta cookie se agrega al encabezado de respuesta HTTP y se envía al cliente como parte de la respuesta.

Al hacer clic en **Enviar**, se ocultan los elementos `ui:label`, `ui:radioButtonGroup` y `ui:button` que se utilizan para seleccionar un lenguaje, y se muestran un elemento `ui:staticText` y dos elementos `ui:hyperlink`. Al principio, cada elemento `ui:staticText` y `ui:hyperlink` tiene establecida su propiedad `rendered` en `false` (líneas 31, 37 y 43). Esto indica que estos componentes no son visibles la primera vez que se carga la página, ya que queremos que la primera vez que el usuario vea la página sólo se incluyan los componentes para seleccionar un lenguaje de programación y enviar la selección.

```
1  <?xml version = "1.0" encoding = "UTF-8"?>
2
3  <!-- Fig. 26.20: Opciones.jsp -->
4  <!-- Archivo JSP que permite al usuario seleccionar un lenguaje de programación -->
5  <jsp:root version = "1.2" xmlns:f = "http://java.sun.com/jsf/core"
6     xmlns:h = "http://java.sun.com/jsf/html" xmlns:jsp =
7     "http://java.sun.com/JSP/Page" xmlns:ui = "http://www.sun.com/web/ui">
8     <jsp:directive.page contentType = "text/html; charset = UTF-8"
9        pageEncoding = "UTF-8"/>
10    <f:view>
11       <ui:page binding = "#{Opciones.page}" id = "page">
12          <ui:html binding = "#{Opciones.html}" id = "html">
13             <ui:head binding = "#{Opciones.head}" id = "head" title=
14                “Opciones”>
15                <ui:link binding = "#{Opciones.link}" id = "link"
16                   url = "/resources/stylesheet.css"/>
17             </ui:head>
18             <ui:body binding = "#{Opciones.body}" id = "body"
19                style = "-rave-layout: grid”>
20                <ui:form binding = "#{Opciones.form}" id = "form">
21                   <ui:label binding = "#{Opciones.etiquetaLenguaje}" for =
22                      "listaLenguajes" id = "etiquetaLenguaje" style =
23                      "font-size: 16px; font-weight: bold; left: 24px; top:
24                      24px; position: absolute" text = "Seleccione un
25                      lenguaje de programación:"/>
26                   <ui:radioButtonGroup binding = "#{Opciones.listaLenguajes}"
27                      id = "listaLenguajes" items =
28                      "#{Opciones.listaLenguajesDefaultOptions.options}" style =
29                      "left: 24px; top: 48px; position: absolute"/>
30                   <ui:staticText binding = "#{Opciones.etiquetaRespuesta}" id =
31                      "etiquetaRespuesta" rendered = "false" style =
32                      "font-size: 16px; font-weight: bold; height: 24px;
33                      left: 24px; top: 24px; position: absolute;
34                      width: 216px"/>
35                   <ui:hyperlink action = "#{Opciones.vinculoLenguajes_action}"
36                      binding = "#{Opciones.vinculoLenguajes}" id =
37                      "vinculoLenguajes" rendered = "false" style = "left:
38                      24px; top: 96px; position: absolute" text = "Haga clic aqui
39                      para elegir otro lenguaje."/>
40                   <ui:hyperlink action =
41                      "#{Opciones.vinculoRecomendaciones_action}" binding =
42                      "#{Opciones.vinculoRecomendaciones}" id =
```

Figura 26.20 | Archivo JSP que permite al usuario seleccionar un lenguaje de programación. (Parte 1 de 3).

```
                     "vinculoRecomendaciones" rendered = "false" style =
                     "left: 24px; top: 120px; position: absolute" text =
                     "Haga clic aqui para obtener recomendaciones de libros."
                     url = "/faces/Recomendaciones.jsp"/>
                  <ui:button action = "#{Opciones.enviar_action}" binding =
                     "#{Opciones.enviar}" id = "enviar" style = "left:
                     23px; top: 192px; position: absolute" text =
                     "Enviar"/>
               </ui:form>
            </ui:body>
         </ui:html>
      </ui:page>
   </f:view>
</jsp:root>
```

a)



b)



Figura 26.20 | Archivo JSP que permite al usuario seleccionar un lenguaje de programación. (Parte 2 de 3).

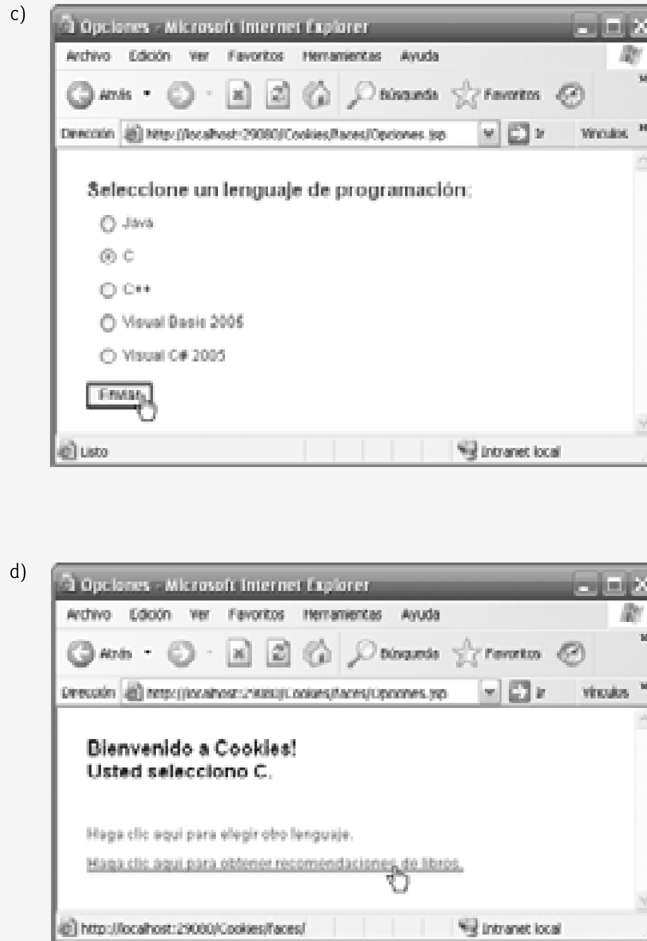


Figura 26.20 | Archivo JSP que permite al usuario seleccionar un lenguaje de programación. (Parte 3 de 3).

El primer hipervínculo (líneas 35 a 39) solicita esta página, y el segundo (líneas 40 a 46) solicita `Recomendaciones.jsp`. La propiedad `url` no se establece para el primer vínculo; hablaremos sobre esto en unos momentos. La propiedad `url` del segundo vínculo se establece en `/faces/Recomendaciones.jsp`. Recuerde que en un ejemplo anterior en este capítulo, establecimos una propiedad `url` a un sitio Web remoto (`http://www.deitel.com`). Para establecer esta propiedad a una página dentro de la aplicación actual, haga clic en el botón de elipsis que está enseguida de la propiedad `url` en la ventana **Propiedades**, para abrir un cuadro de diálogo. Use este cuadro de diálogo para seleccionar una página dentro de su proyecto como destino para el vínculo.

Cómo agregar una nueva página y crear un vínculo

Para establecer la propiedad `url` a una página en la aplicación actual, la página de destino debe existir de antemano. Para establecer la propiedad `url` de un vínculo a `Recomendaciones.jsp`, primero debemos crear esta página. Haga clic en el nodo **Páginas Web** en la ventana **Propiedades** y seleccione **Nuevo > Página** en el menú que aparezca. En el cuadro de diálogo **Nuevo Página**, cambie el nombre de la página a `Recomendaciones` y haga clic en **Terminar** para crear los archivos `Recomendaciones.jsp` y `Recomendaciones.java`. (En breve hablaremos sobre el contenido de estos archivos). Una vez que exista el archivo `Recomendaciones.jsp`, puede seleccionarlo como el valor de `url` para `vinculoRecomendaciones`.

Para `Opciones.jsp`, en vez de establecer la propiedad `url` de `vinculoLenguajes`, vamos a agregar al bean de página un manejador de acciones para este componente. El manejador de acciones nos permitirá mostrar y

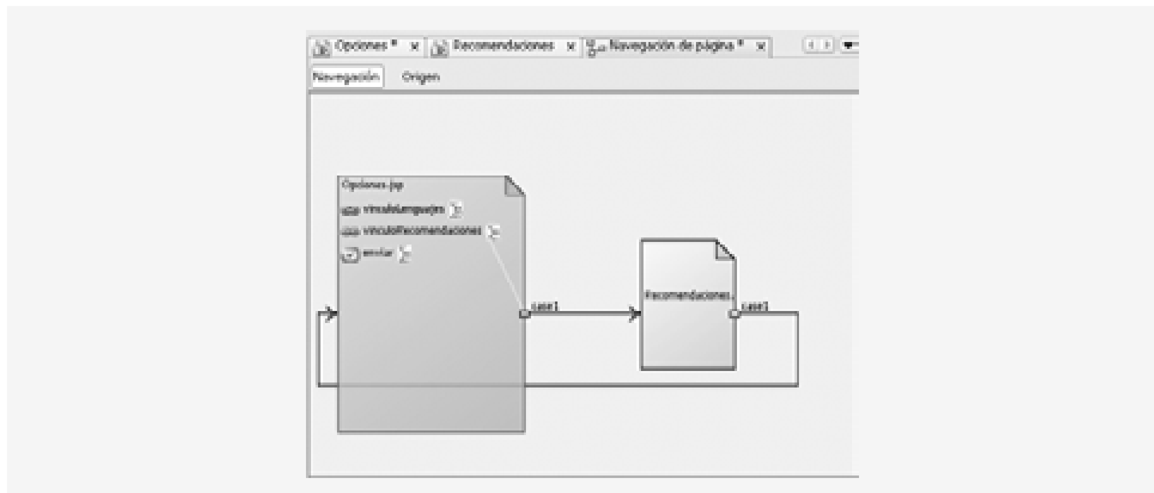


Figura 26.21 | Edición del archivo **Navegación de página**.

ocultar componentes de la página, sin necesidad de redirigir al usuario a otra página. Si especificamos un `url` de destino, se sobrescribirá el manejador de acciones del componente y el usuario será redirigido a la página especificada, por lo cual es importante que no establezcamos la propiedad `url` en este caso. Como vamos a utilizar este vínculo para volver a cargar la página actual, el manejador de acciones simplemente debe devolver `null`, lo cual hará que `Opciones.jsp` se vuelva a cargar.

Para agregar un manejador de acciones a un hipervínculo que también debe dirigir al usuario a otra página, debemos agregar una regla al archivo **Navegación de página** (figura 26.21). Para editar este archivo, haga clic con el botón derecho del ratón en cualquier parte del Diseñador visual y seleccione **Navegación de página...**. Localice el vínculo cuya regla de navegación desea establecer y arrástrelo a la página de destino. Ahora el vínculo puede dirigir al usuario a una nueva página sin sobrescribir su manejador de acciones. También es útil editar el archivo **Navegación de página** cuando es conveniente tener elementos de acción que no puedan especificar una propiedad `url`, como los botones, para dirigir a los usuarios a otra página.

La figura 26.22 contiene el código que escribe una cookie al equipo cliente, cuando el usuario selecciona un lenguaje de programación. El archivo también determina cuáles componentes deben aparecer en la página, mostrando ya sea los componentes para elegir un lenguaje, o los vínculos para navegar por la aplicación, dependiendo de las acciones del usuario.

```
// Fig. 26.22: Opciones.java
// Bean de página que almacena la selección de lenguaje del usuario como
// una cookie en el cliente.
package cookies;

import com.sun.rave.web.ui.appbase.AbstractPageBean;
import com.sun.rave.web.ui.component.Body;
import com.sun.rave.web.ui.component.Form;
import com.sun.rave.web.ui.component.Head;
import com.sun.rave.web.ui.component.Html;
import com.sun.rave.web.ui.component.Link;
import com.sun.rave.web.ui.component.Page;
import javax.faces.FacesException;
import com.sun.rave.web.ui.component.StaticText;
```

Figura 26.22 | Bean de página que almacena la selección de lenguaje del usuario como una cookie en el cliente. (Parte 1 de 4).

```java
import com.sun.rave.web.ui.component.Label;
import com.sun.rave.web.ui.component.RadioButtonGroup;
import com.sun.rave.web.ui.model.SingleSelectOptionsList;
import com.sun.rave.web.ui.component.Hyperlink;
import com.sun.rave.web.ui.component.Button;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.Cookie;
import java.util.Properties;

public class Opciones extends AbstractPageBean
{
   private int __placeholder;

   // el método _init inicializa los componentes y establece
   // las opciones para el grupo de botones de selección.
   private void _init() throws Exception
   {
      listaLenguajesDefaultOptions.setOptions(
         new com.sun.rave.web.ui.model.Option[]
         {
            new com.sun.rave.web.ui.model.Option("Java", "Java"),
            new com.sun.rave.web.ui.model.Option("C", "C"),
            new com.sun.rave.web.ui.model.Option("C++", "C++"),
            new com.sun.rave.web.ui.model.Option("Visual/Basic/2005",
               "Visual Basic 2005"),
            new com.sun.rave.web.ui.model.Option("Visual/C#/2005",
              "Visual C# 2005")
         }
      );
   } // fin del método _init

   // Para ahorrar espacio, omitimos el código en las líneas 46 a 203. El código
   // fuente completo se proporciona con los ejemplos de este capítulo.

   private Properties libros = new Properties();

   // Construye una nueva instancia del bean de página e inicializa las propiedades
   // que asocian los lenguajes con los números ISBN de los libros recomendados.
   public Opciones()
   {
      // inicializa el objeto Properties de los valores que se van
      // a almacenar como cookies.
      libros.setProperty( "Java", "0-13-222220-5" );
      libros.setProperty( "C", "0-13-142644-3" );
      libros.setProperty( "C++", "0-13-185757-6" );
      libros.setProperty( "Visual/Basic/2005", "0-13-186900-0" );
      libros.setProperty( "Visual/C#/2005", "0-13-152523-9" );
   } // fin del constructor de Opciones

   protected ApplicationBean getApplicationBean()
   {
      return (ApplicationBean) getBean( "ApplicationBean" );
   } // fin del método getApplicationBean

   protected RequestBean getRequestBean()
   {
      return (RequestBean) getBean( "RequestBean" );
   } // fin del método getRequestBean
```

Figura 26.22 | Bean de página que almacena la selección de lenguaje del usuario como una cookie en el cliente. (Parte 2 de 4).

```

   protected SessionBean getSessionBean()
   {
      return (SessionBean) getBean( "SessionBean" );
   } // fin del método getSessionBean

   public void init()
   {
      super.init();
      try
      {
         _init();
      } // fin de try
      catch ( Exception e )
      {
         log( "Error al inicializar Opciones", e );
         throw e instanceof FacesException ? ( FacesException ) e:
            new FacesException( e );
      } // fin de catch
   } // fin del método init

   public void preprocess()
   {
      // cuerpo vacío
   } // fin del método preprocess

   public void prerender()
   {
      // cuerpo vacío
   } // fin del método prerender

   public void destroy()
   {
      // cuerpo vacío
   } // fin del método destroy

   // Manejador de acciones para el botón Enviar. Verifica si se seleccionó un
   // lenguaje y, de ser así, registra una cookie para ese lenguaje, y
   // establece la etiquetaRespuesta para indicar el lenguaje seleccionado.
   public String enviar_action()
   {
      String msj = "Bienvenido a Cookies!   Usted ";

      // si el usuario hizo una selección
      if ( listaLenguajes.getSelected() != null )
      {
         String lenguaje = listaLenguajes.getSelected().toString();
         String mostrarLenguaje = lenguaje.replace( '/', ' ' );
         msj += "selecciono " + mostrarLenguaje + ".";

         // obtiene el número ISBN del libro para el lenguaje dado.
         String ISBN = libros.getProperty( lenguaje );

         // crea cookie usando un par nombre-valor de lenguaje-ISBN
         Cookie cookie = new Cookie( lenguaje, ISBN );

         // agrega la cookie al encabezado de respuesta para colocarla en
         // el equipo del usuario
```

Figura 26.22 | Bean de página que almacena la selección de lenguaje del usuario como una cookie en el cliente. (Parte 3 de 4).

```
286             HttpServletResponse respuesta =
287                (HttpServletResponse) getExternalContext().getResponse();
288             respuesta.addCookie( cookie );
289          } // fin de if
290          else
291             msj += "no selecciono un lenguaje.";
292
293          etiquetaRespuesta.setValue(msj);
294          listaLenguajes.setRendered(false);
295          etiquetaLenguaje.setRendered(false);
296          enviar.setRendered(false);
297          etiquetaRespuesta.setRendered(true);
298          vinculoLenguajes.setRendered(true);
299          vinculoRecomendaciones.setRendered(true);
300          return null; // vuelve a cargar la página
301       } // fin del método enviar_action
302
303       // vuelve a mostrar los componentes utilizados para permitir al usuario
304       // seleccionar un lenguaje.
305       public String vinculoLenguajes_action()
306       {
307          etiquetaRespuesta.setRendered(false);
308          vinculoLenguajes.setRendered(false);
309          vinculoRecomendaciones.setRendered(false);
310          listaLenguajes.setRendered(true);
311          etiquetaLenguaje.setRendered(true);
312          enviar.setRendered(true);
313           return null;
314       } // fin del método vinculoLenguajes_action
315    } // fin de class Opciones
```

Figura 26.22 | Bean de página que almacena la selección de lenguaje del usuario como una cookie en el cliente. (Parte 4 de 4).

Como dijimos antes, el método `_init` maneja la inicialización de componentes. Como esta página contiene un objeto `RadioButtonGroup` que requiere inicialización, el método `_init` (líneas 30 a 44) construye un arreglo de objeto `Options` que van a mostrar los botones. En las líneas 38 y 40, los nombres de las opciones contienen barras diagonales en vez de espacios, ya que posteriormente los utilizamos como nombres de cookies y Java no permite que los nombres de cookies tengan espacios.

En las líneas 212 a 216 del constructor se inicializa un objeto `Properties`: una estructura de datos que almacena pares clave-valor tipo `String`. La aplicación utiliza la clave para almacenar y obtener el valor asociado en el objeto `Properties`. En este ejemplo, las claves son objetos `String` que contienen los nombres de los lenguajes de programación, y los valores son objetos `String` que contienen los números ISBN para los libros recomendados. La clase `Properties` proporciona el método `setProperty`, el cual recibe como argumentos una clave y un valor. Un valor que se agrega a través del método `setProperty` se coloca en el objeto `Properties`, en una ubicación determinada por la clave. El valor para una entrada específica en el objeto `Properties` se puede determinar invocando al método `getProperty` en el objeto `Properties`, con la clave del valor como argumento.



Observación de ingeniería de software 26.1

Java Studio Creator2 puede importar automáticamente cualquier paquete que necesite su archivo de Java y que no esté incluido. Por ejemplo, después de agregar el objeto `Properties` *a* `Opciones.java`*, puede hacer clic con el botón derecho en la ventana del editor de Java y seleccionar* **Corregir importaciones** *para importar automáticamente el paquete* `java.util.Properties`*.*

Al hacer clic en **Enviar**, se invoca el manejador de eventos `enviar_action` (líneas 267 a 301), el cual muestra un mensaje indicando el lenguaje seleccionado en el elemento `etiquetaRespuesta`, y agrega una nueva cookie a

la respuesta. Si se seleccionó un lenguaje (línea 272) se obtiene el valor seleccionado (línea 274). En la línea 275 se convierte la selección en un objeto `String` que se puede mostrar en la `etiquetaRespuesta`, sustituyendo las barras diagonales con espacios. En la línea 276 se agrega el lenguaje seleccionado al mensaje de resultados.

En la línea 279 se obtiene el ISBN para el lenguaje seleccionado de las propiedades (`Properties`) de los libros. Después, en la línea 282 se crea un nuevo objeto cookie (de la clase `Cookie` en el paquete `javax.servlet.http`), usando el lenguaje seleccionado como el nombre de la cookie y su correspondiente número ISBN como el valor de la cookie. Esta cookie se agrega al encabezado de respuesta HTTP en las líneas 286 a 288. Un objeto de la clase `HttpServletResponse` (del paquete `javax.servlet.http`) representa la respuesta. Para acceder a este objeto, se invoca el método `getExternalContext` en el bean de página y después se invoca a `getResponse` en el objeto resultante. Si no se seleccionó un lenguaje, en la línea 291 se establece el mensaje de resultados para indicar que no hubo selección.

Las líneas 293 a 299 controlan la apariencia de la página, una vez que el usuario hace clic en **Enviar**. En la línea 293 se establece la `etiquetaRespuesta` para mostrar el objeto `String` llamado `msj`. Como el usuario acaba de enviar una selección de lenguaje, se ocultan los componentes utilizados para recolectar la selección (líneas 294 a 296), y se muestran `etiquetaRespuesta` y los vínculos utilizados para navegar por la aplicación (líneas 297 a 299). El manejador de acciones devuelve `null` en la línea 300, la cual vuelve a cargar `Opciones.jsp`.

Las líneas 305 a 311 contienen el manejador de eventos de `vinculoLenguajes`. Cuando el usuario hace clic en este vínculo, se ocultan `etiquetaRespuesta` y los dos vínculos (líneas 307 y 309), y se vuelven a mostrar los componentes que permiten al usuario seleccionar un lenguaje (líneas 310 a 312). El método devuelve `null` en la línea 313, lo cual hace que `Opciones.jsp` se vuelva a cargar.

Cómo mostrar las recomendaciones de libros con base en los valores de cookies

Después de hacer clic en **Enviar**, el usuario puede solicitar la recomendación de un libro. El hipervínculo de recomendaciones de libros lleva al usuario a `Recomendaciones.jsp` (figura 26.23) para mostrar las recomendaciones con base en los lenguajes seleccionados por el usuario.

```
<?xml version = "1.0" encoding = "UTF-8"?>

<!-- Fig. 26.23: Recomendaciones.jsp -->
<!-- Muestra las recomendaciones de libros mediante el uso de cookies -->
<jsp:root version = "1.2" xmlns:f = "http://java.sun.com/jsf/core"
   xmlns:h = "http://java.sun.com/jsf/html" xmlns:jsp =
   "http://java.sun.com/JSP/Page" xmlns:ui = "http://www.sun.com/web/ui">
   <jsp:directive.page contentType = "text/html; charset = UTF-8"
      pageEncoding = "UTF-8"/>
   <f:view>
      <ui:page binding = "#{Recomendaciones.page}" id = "page">
         <ui:html binding = "#{Recomendaciones.html}" id = "html">
            <ui:head binding = #{Recomendaciones.head}" id = "head"
               title = "Recomendaciones">
               <ui:link binding = "#{Recomendaciones.link}" id = "link"
               url= "/resources/stylesheet.css"/>
            </ui:head>
            <ui:body binding = "#{Recomendaciones.body}" id = "body"
               style = "-rave-layout: grid">
               <ui:form binding = "#{Recomendaciones.form}" id = "form">
                  <ui:label binding = "#{Recomendaciones.etiquetaLenguaje}"
                     for = "cuadroListaLibrosOpciones" id = "etiquetaLenguaje" style =
                     "font-size: 20px; font-weight: bold; left: 24px; top:
                     24px; position: absolute" text = "Recomendaciones"/>
                  <ui:listbox binding = "#{Recomendaciones.cuadroListaLibros}"
                     id = "cuadroListaLibros" items = "#{Recomendaciones.
                     cuadroListaLibrosOpciones.options}" rows = "6" style =
                     "left: 24px; top: 72px; position: absolute;
```

Figura 26.23 | Archivo de JSP que muestra las recomendaciones de libros, con base en cookies. (Parte 1 de 2).

```
29                         width: 360px"/>
30                      <ui:hyperlink action = "case1" binding =
31                         "#{Recomendaciones.vinculoOpciones}" id = "vinculoOpciones"
32                         style = "left: 24px; top: 192px; position: absolute"
33                         text = "Haga clic aquí para elegir otro lenguaje."/>
34                   </ui:form>
35                </ui:body>
36             </ui:html>
37          </ui:page>
38       </f:view>
39    </jsp:root>
```

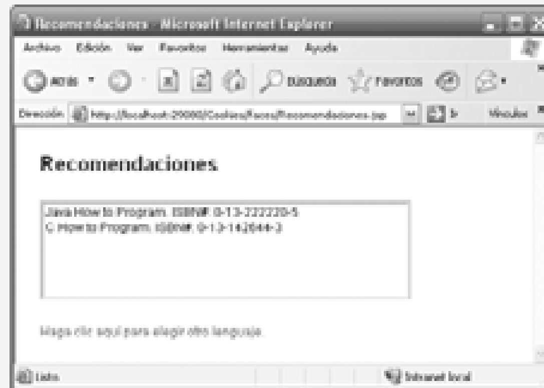


Figura 26.23 | Archivo de JSP que muestra las recomendaciones de libros, con base en cookies. (Parte 2 de 2).

`Recomendaciones.jsp` contiene un componente **Etiqueta** (líneas 21 a 24), un **List box** (líneas 25 a 29) y un **Hipervínculo** (líneas 30 a 33). La **Etiqueta** muestra el texto `Recomendaciones` en la parte superior de la página. Un componente **List box** muestra una lista de opciones, de las que el usuario puede seleccionar varias. El **List box** en este ejemplo muestra las recomendaciones creadas por el bean de página `Recomendaciones.java` (figura 26.24), o el texto `"No hay recomendaciones. Por favor seleccione un lenguaje"`. El **Hipervínculo** permite al usuario regresar a `Opciones.jsp` para seleccionar más lenguajes.

Bean de página que crea las recomendaciones de libros a partir de cookies

En `Recomendaciones.java` (figura 26.24), el método `prerender` (líneas 192 a 223) obtiene las cookies del cliente, usando el método `getCookies` del objeto petición (líneas 195 a 197). Un objeto de la clase `HttpServletRequest` (del paquete `javax.servlet.http`) representa la petición. Este objeto puede obtenerse invocando al método `getExternalContext` en el bean de página, y después invocando a `getRequest` en el objeto resultante. La llamada a `getCookies` devuelve un arreglo de las cookies que se habían escrito antes en el cliente. Una aplicación puede leer las cookies sólo si un servidor las creó en el dominio en el que se ejecuta la aplicación; un servidor Web no puede acceder a las cookies creadas por los servidores en otros dominios. Por ejemplo, una cookie creada por un servidor Web en el dominio `deitel.com` no puede ser leída por un servidor Web de cualquier otro dominio.

En la línea 203 se determina si por lo menos existe una cookie. (Ignoramos la primera cookie en el arreglo, ya que contienen información que no es específica para nuestra aplicación). En las líneas 205 a 213 se agrega la información en la(s) cookie(s) a un arreglo de tipo `Option`. Los arreglos de objetos `Option` pueden mostrarse como una lista de elementos en un componente **List box**. El ciclo obtiene el nombre y el valor de cada cookie, usando la variable de control para determinar el valor actual en el arreglo de cookies. Si no se seleccionó un lenguaje, en las líneas 215 a 220 se agrega un mensaje a un arreglo de tipo `Option`, el cual pide al usuario que seleccione un lenguaje. En la línea 222 se establece `cuadroListaLibros` para mostrar el arreglo de tipo `Option` resultante. En la figura 26.25 sintetizamos los métodos de `Cookie` de uso común.

```java
// Fig. 26.24: Recomendaciones.java
// Bean de pagina que muestra las recomendaciones de libros con base en cookies
// que almacenan los lenguajes de programación seleccionados por el usuario.
package cookies;

import com.sun.rave.web.ui.appbase.AbstractPageBean;
import com.sun.rave.web.ui.component.Body;
import com.sun.rave.web.ui.component.Form;
import com.sun.rave.web.ui.component.Head;
import com.sun.rave.web.ui.component.Html;
import com.sun.rave.web.ui.component.Link;
import com.sun.rave.web.ui.component.Page;
import javax.faces.FacesException;
import com.sun.rave.web.ui.component.Listbox;
import com.sun.rave.web.ui.model.DefaultOptionsList;
import com.sun.rave.web.ui.component.Label;
import com.sun.rave.web.ui.component.Hyperlink;
import com.sun.rave.web.ui.model.Option;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.Cookie;
import com.sun.rave.web.ui.component.HiddenField;

public class Recomendaciones extends AbstractPageBean
{
   private int __placeholder;

   private void _init() throws Exception
   {
      // cuerpo vacío
   } // fin del método _init()

   // Para ahorrar espacio, omitimos el código en las líneas 32 a 151. El código
   // fuente completo se proporciona con los ejemplos de este capítulo.

      public Recomendaciones()
      {
         // constructor vacío
      } // fin del constructor

      protected ApplicationBean getApplicationBean()
      {
         return (ApplicationBean) getBean( "ApplicationBean" );
      } // fin del método getApplicationBean

      protected RequestBean getRequestBean()
      {
         return (RequestBean) getBean( "RequestBean" );
      } // fin del método getRequestBean

      protected SessionBean getSessionBean()
      {
         return (SessionBean) getBean( "SessionBean" );
      } // fin del método getSessionBean

      public void init()
      {
         super.init();
         try
```

Figura 26.24 | Bean de página que muestra las recomendaciones de libros con base en cookies que almacenan los lenguajes de programación seleccionados por el usuario. (Parte 1 de 2).

```
         {
            _init();
         } // fin de try
         catch ( Exception e )
         {
            log( "Error al inicializar Recomendaciones", e );
            throw e instanceof FacesException ? (FacesException) e:
               new FacesException( e );
         } // fin de catch
      } // fin del método init

      public void preprocess()
      {
         // cuerpo vacío
      } // fin del método preprocess

      public void prerender()
      {
         // obtiene las cookies del cliente
         HttpServletRequest peticion =
            (HttpServletRequest)getExternalContext().getRequest();
         Cookie [] cookies = peticion.getCookies();

         // si hay cookies, almacena los correspondientes libros y
         // números ISBN en un arreglo de Opciones
         Option [] recomendaciones;

         if ( cookies.length > 1 )
         {
            recomendaciones = new Option[ cookies.length - 1 ];
            for ( int i = 0; i < cookies.length - 1; i++ )
            {
               String lenguaje =
                  cookies[i].getName().replace( '/', ' ' );
               recomendaciones[ i ] = new Option( lenguaje +
                  "How to Program. ISBN#: " + cookies[i].getValue() );
            } // fin de for
         } // fin de if

         // en caso contrario, almacena un mensaje indicando que no se seleccionó un
         lenguaje
         else
         {
            recomendaciones = new Option[ 1 ];
            recomendaciones[ 0 ] = new Option(
               "No hay recomendaciones. " + "Seleccione un lenguaje." ) ;
         } // fin de else

         cuadroListaLibros.setItems(recomendaciones);
      } // fin del método prerender

      public void destroy()
      {
         // cuerpo vacío
      } // fin del método destroy
   } // fin de la clase Recomendaciones
```

Figura 26.24 | Bean de página que muestra las recomendaciones de libros con base en cookies que almacenan los lenguajes de programación seleccionados por el usuario. (Parte 2 de 2).

Métodos	Descripción
`getDomain`	Devuelve un objeto `String` que contiene el dominio de la cookie (es decir, el dominio desde el que se escribió la cookie). Esto determina cuáles servidores Web pueden recibir la cookie. De manera predeterminada, las cookies se envían al servidor Web que envió originalmente la cookie al cliente. Si se modifica la propiedad `Domain`, la cookie se devolverá a un servidor Web distinto del que la escribió originalmente.
`getMaxAge`	Devuelve un valor `int`, el cual indica cuántos segundos persistirá la cookie en el navegador. El valor predeterminado es -1, lo cual significa que la cookie persistirá hasta que el navegador se cierre.
`getName`	Devuelve un objeto `String` que contiene el nombre de la cookie.
`getPath`	Devuelve un objeto `String` que contiene la ruta a un directorio en el servidor, en el cual se aplica la cookie. Las cookies pueden "dirigirse" a directorios específicos en el servidor Web. De manera predeterminada, una cookie se devuelve sólo a las aplicaciones que operan en el mismo directorio que la aplicación que envío la cookie, o en un subdirectorio de ese directorio. Si se modifica la propiedad `Path`, la cookie se devolverá a un directorio distinto al directorio en el que se escribió originalmente.
`getSecure`	Devuelve un valor `bool` que indica si la cookie debe transmitirse a través de un protocolo seguro. El valor `true` hace que se utilice un protocolo seguro.
`getValue`	Devuelve un objeto `String` que contiene el valor de la cookie.

Figura 26.25 | Métodos de `javax.servlet.http.Cookie`.

26.7.2 Rastreo de sesiones con el objeto SessionBean

También podemos realizar el rastreo de sesiones mediante la clase `SessionBean` que se proporciona en cada una de las aplicaciones Web creadas con Java Studio Creator 2. Cuando se solicita una página Web que está dentro del proyecto, se crea un objeto `SessionBean`. Se puede acceder a las propiedades de este objeto durante una sesión con el navegador, mediante la invocación del método `getSessionBean` en el bean de página. Para demostrar las técnicas de rastreo de sesiones usando un objeto `SessionBean`, modificamos los archivos de bean de página de las figuras 26.22 y 26.24, de manera que utilicen el objeto `SessionBean` para almacenar los lenguajes seleccionados por el usuario. Empezamos con el archivo `Opciones.jsp` actualizado (figura 26.26). La figura 26.29 presenta el archivo `SessionBean.java` y la figura 26.30 presenta el archivo de bean de página modificado para `Opciones.jsp.`

El archivo `Opciones.jsp` de la figura 26.26 es similar al que se presenta en la figura 26.20 para el ejemplo de las cookies. En las líneas 38 a 45 se definen dos elementos `ui:staticText` que no se presentaron en el ejemplo de las cookies. El primer elemento muestra el texto `"Número de selecciones hasta ahora:"`. El atributo `text` del segundo elemento está enlazado a la propiedad `numSelecciones` en el objeto `SessionBean` (líneas 44 y 45). En un momento hablaremos acerca de cómo enlazar el atributo `text` a una propiedad de `SessionBean`.

```
1  <?xml version = "1.0" encoding = "UTF-8"?>
2
3  <!-- Fig. 26.26: Opciones.jsp -->
4  <!-- Archivo JSP que permite al usuario seleccionar un lenguaje de programación. -->
5  <jsp:root version = "1.2" xmlns:f = "http://java.sun.com/jsf/core"
6     xmlns:h = "http://java.sun.com/jsf/html" xmlns:jsp =
```

Figura 26.26 | Archivo JSP que permite al usuario seleccionar un lenguaje de programación. (Parte 1 de 4).

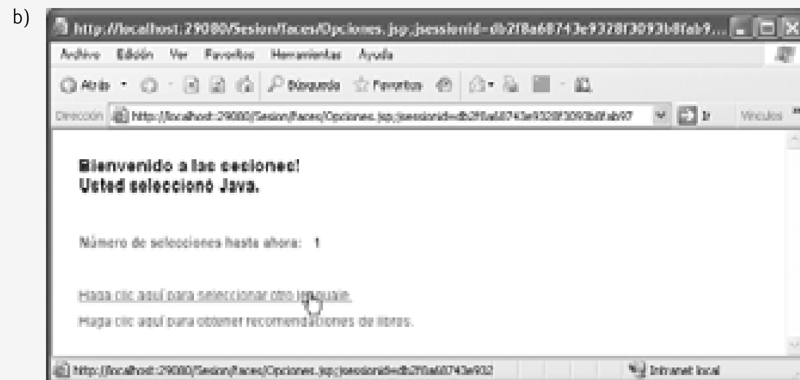
```
    "http://java.sun.com/JSP/Page" xmlns:ui = "http://www.sun.com/web/ui">
    <jsp:directive.page contentType = "text/html; charset = UTF-8"
       pageEncoding = "UTF-8"/>
    <f:view>
       <ui:page binding = "#{Opciones.page}" id = "page">
          <ui:html binding = "#{Opciones.html}" id = "html">
             <ui:head binding = "#{Opciones.head}" id = "head">
                <ui:link binding = "#{Opciones.link}" id = "link"
                   url = "/resources/stylesheet.css"/>
                </ui:head>
                <ui:body binding = "#{Opciones.body}" id = "body"
                   style = "-rave-layout: grid">
                   <ui:form binding = "#{Opciones.form}" id = "form">
                      <ui:label binding = "#{Opciones.etiquetaLenguaje}" for =
                         "listaLenguajes" id = "etiquetaLenguaje" style =
                         "font-size: 16px; font-weight: bold; left: 24px; top:
                         24px; position: absolute" text = "Seleccione un
                         lenguaje de programación:"/>
                      <ui:radioButtonGroup binding = "#{Opciones.listaLenguajes}"
                         id = "listaLenguajes" items =
                         "#{Opciones.listaLenguajesDefaultOptions.options}" style =
                         "left: 24px; top: 48px; position: absolute"/>
                      <ui:button action = "#{Opciones.enviar_action}" binding =
                         "#{Opciones.enviar}" id = "enviar" style = "left:
                         23px; top: 192px; position: absolute" text =
                         "Enviar"/>
                      <ui:staticText binding = "#{Opciones.etiquetaRespuesta}" id =
                         "etiquetaRespuesta" rendered = "false" style =
                         "font-size: 16px; font-weight: bold; height: 24px;
                         left: 24px; top: 24px; position: absolute;
                         width: 216px"/>
                      <ui:staticText binding = "#{Opciones.etiquetaNumSelec}"
                         id = "etiquetaNumSelec" rendered = "false" style =
                         "left: 24px; top: 96px; position: absolute" text =
                         "Número de selecciones hasta ahora:"/>
                      <ui:staticText binding = "#{Opciones.numSelec}" id =
                         "numSelec" rendered = "false" style = "left:
                         240px; top: 96px; position: absolute" text =
                         "#{SessionBean1.numSelecciones}"/>
                   <ui:hyperlink action = "case1" binding = "#{Opciones.
                   vinculoRecomendaciones}"
                         id = "vinculoRecomendaciones" rendered = "false" style = "left:
                         24px; top: 168px; position: absolute" text =
                         "Haga clic aquí para obtener recomendaciones de libros." url =
                         "/faces/Recomendaciones.jsp"/>
                      <ui:hyperlink action = "#{Opciones.vinculoLenguajes1_action}"
                         binding = "#{Opciones.vinculoLenguajes1}" id =
                         "vinculoLenguajes1" rendered = "false" style = "left:
                         24px; top: 144px; position: absolute" text = "Haga clic
                         aquí para seleccionar otro lenguaje."/>
                   </ui:form>
                </ui:body>
          </ui:html>
       </ui:page>
    </f:view>
</jsp:root>
```

Figura 26.26 | Archivo JSP que permite al usuario seleccionar un lenguaje de programación. (Parte 2 de 4).

a)



b)



c)



Figura 26.26 | Archivo JSP que permite al usuario seleccionar un lenguaje de programación. (Parte 3 de 4).

d)



Figura 26.26 | Archivo JSP que permite al usuario seleccionar un lenguaje de programación. (Parte 4 de 4).

Cómo agregar propiedades al objeto `SessionBean`

En este ejemplo, utilizamos el rastreo de sesiones para almacenar no sólo los lenguajes seleccionados por el usuario, sino también el número de selecciones realizadas. Para almacenar esta información en el objeto `SessionBean`, agregamos propiedades a la clase `SessionBean`.

Para agregar una propiedad que almacene el número de selecciones hasta ahora, haga clic con el botón derecho del ratón en el nodo `SessionBean1` de la ventana **Esquema** y seleccione **Agregar | Propiedad** para que aparezca el cuadro de diálogo **Nuevo patrón de propiedad** (figura 26.27). Este cuadro de diálogo nos permite agregar propiedades primitivas, `String` o de envoltura de tipo primitivo al objeto `SessionBean1`. Agregue una propiedad `int` llamada `numSelecciones` y haga clic en **Aceptar** para aceptar las opciones predeterminadas para esta propiedad. Abra el archivo `SessionBean1` y verá una nueva definición de propiedad, un método *get* y un método *set* para `numSelecciones`.

La propiedad `numSelecciones` se manipulará en el archivo de bean de página para almacenar el número de lenguajes que seleccionó el usuario. Para mostrar el valor de esta propiedad en el elemento **Texto estático** llamado `numSelec` en el archivo JSP, haga clic con el botón derecho en el componente **Texto estático** en la ventana **Esquema** en modo **Diseño**, y seleccione **Enlazar con datos...**. En el cuadro de diálogo **Enlazar con datos** (figura 26.28), seleccione la ficha **Enlazar con un objeto**, localice la propiedad `numSelecciones` bajo el nodo `SessionBean1` y haga clic en **Aceptar**. Ahora, el elemento **Texto estático** mostrará el valor de la propiedad `numSelecciones` de `SessionBean1`. Si cambia el valor de la propiedad el texto también cambia, de manera que no es necesario establecer el texto en el bean de página mediante programación.



Figura 26.27 | Cuadro de diálogo **Nuevo patrón de propiedad** para agregar una propiedad al objeto `SessionBean`.

Ahora que hemos agregado una propiedad para almacenar el número de selecciones en el objeto `SessionBean`, debemos agregar una segunda propiedad para almacenar las selecciones en sí. Nos gustaría almacenar las selecciones con pares clave-valor del lenguaje seleccionado y el número ISBN de un libro relacionado, algo similar a la forma en que se almacenaron las selecciones mediante el uso de cookies. Para hacer esto, agregamos un objeto `Properties` llamado `lenguajesSeleccionados` al objeto `SessionBean`. Agregamos en forma manual esta propiedad al archivo `SessionBean`, pero podemos agregarla usando el cuadro de diálogo **Nuevo patrón de propiedad** en la figura 26.27. Simplemente escriba `java.util.Properties` en el campo del cuadro de lista desplegable **Tipo**, configure la propiedad y haga clic en **Aceptar**. El archivo `SessionBean` final, después de haber agregado las dos propiedades, se muestra en la figura 26.29.



Figura 26.28 | Cuadro de diálogo **Enlazar con datos**.

```
// Fig. 26.29: SessionBean1.java
// Archivo SessionBean1 para almacenar las selecciones de lenguajes.
package sesion;

import com.sun.rave.web.ui.appbase.AbstractSessionBean;
import java.util.Properties;
import javax.faces.FacesException;

public class SessionBean1 extends AbstractSessionBean
{
   private int __placeholder;

   private void _init() throws Exception
   {
      // cuerpo vacío
   } // fin del método _init

   public SessionBean1()
   {
      // constructor vacío
   } // fin del constructor

   protected ApplicationBean1 getApplicationBean1()
```

Figura 26.29 | Archivo `SessionBean1` para almacenar las selecciones de lenguajes. (Parte 1 de 2).

```
    {
       return (ApplicationBean1) getBean( "ApplicationBean1" );
    } // fin del método getApplicationBean1

    public void init()
    {
       super.init();
       try
       {
          _init();
       } // fin de try
       catch ( Exception e )
       {
          log( "Error al inicializar SessionBean1", e);
          throw e instanceof FacesException ? (FacesException) e:
             new FacesException( e );
       } // fin de catch
    } // fin del método init

    public void passivate()
    {
       // cuerpo vacío
    } // fin del método passivate

    public void activate()
    {
       // cuerpo vacío
    } // fin del método activate

    public void destroy()
    {
       // cuerpo vacío
    } // fin del método destroy

    private int numSelecciones = 0; // almacena el número de selecciones únicas

    public int getNumSelecciones()
    {
       return this.numSelecciones;
    } // fin del método getNumSelecciones

    public void setNumSelecciones( int numSelecciones )
    {
       this.numSelecciones = numSelecciones;
    } // fin del método setNumSelecciones

    // Almacena pares clave-valor de lenguajes seleccionados
    private Properties lenguajesSeleccionados = new Properties();

    public Properties getLenguajesSeleccionados()
    {
       return this.lenguajesSeleccionados;
    } // fin del método getLenguajesSeleccionados

    public void setLenguajesSeleccionados( Properties lenguajesSeleccionados )
    {
       this.lenguajesSeleccionados = lenguajesSeleccionados;
    } // fin del método setLenguajesSeleccionados
 } // fin de la clase SessionBean1
```

Figura 26.29 | Archivo `SessionBean1` para almacenar los lenguajes seleccionados. (Parte 2 de 2).

En la línea 58 se declara la propiedad `numSelecciones`, y en las líneas 60 a 63 y 65 a 68 se declaran sus métodos *obtener (get)* y *establecer (set)*, respectivamente. Esta parte del código se generó de manera automática cuando utilizamos el cuadro de diálogo **Nuevo patrón de propiedad**. En la línea 71 se define el objeto `Properties llamado lenguajesSeleccionados` que almacenará las selecciones del usuario. En las líneas 73 a 76 y 78 a 81 están los métodos *get* y *set* para esta propiedad.

Manipulación de las propiedades de `SessionBean` *en un archivo de bean de página*

El archivo de bean de página para la página `Opciones.jsp` se muestra en la figura 26.30. Debido a que gran parte de este ejemplo es idéntica al anterior, nos concentraremos en las nuevas características.

```
// Fig. 26.30: Opciones.java
// Bean de página que almacena las selecciones de lenguajes en una propiedad SessionBean.
package sesion;

import com.sun.rave.web.ui.appbase.AbstractPageBean;
import com.sun.rave.web.ui.component.Body;
import com.sun.rave.web.ui.component.Form;
import com.sun.rave.web.ui.component.Head;
import com.sun.rave.web.ui.component.Html;
import com.sun.rave.web.ui.component.Link;
import com.sun.rave.web.ui.component.Page;
import javax.faces.FacesException;
import com.sun.rave.web.ui.component.RadioButtonGroup;
import com.sun.rave.web.ui.component.Hyperlink;
import com.sun.rave.web.ui.component.Button;
import com.sun.rave.web.ui.component.Label;
import com.sun.rave.web.ui.component.StaticText;
import com.sun.rave.web.ui.model.SingleSelectOptionsList;
import java.util.Properties;
import javax.servlet.http.Cookie;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpSession;

public class Opciones extends AbstractPageBean
{
   private int __placeholder;

   private void _init() throws Exception
   {
      listaLenguajesDefaultOptions.setOptions(
         new com.sun.rave.web.ui.model.Option[]
         {
            new com.sun.rave.web.ui.model.Option( "Java", "Java" ),
            new com.sun.rave.web.ui.model.Option( "C", "C" ),
            new com.sun.rave.web.ui.model.Option( "C++", "C++" ),
            new com.sun.rave.web.ui.model.Option( "Visual Basic 2005",
               "Visual Basic 2005" ),
            new com.sun.rave.web.ui.model.Option( "Visual C# 2005”,
               "Visual C# 2005" )
         }
      );
   } // fin del método init

   // para ahorrar espacio, omitimos el código de las líneas 44 a 219. El código
   // fuente completo se proporciona con los ejemplos de este capítulo.
```

Figura 26.30 | Bean de página que almacena las selecciones de lenguajes en una propiedad `SessionBean`. (Parte 1 de 3).

```

    private Properties libros = new Properties();

    public Opciones()
    {
       // inicializa el objeto Properties de valores que se van a almacenar en
       // el bean de sesión.
       libros.setProperty( "Java", "0-13-222220-5" );
       libros.setProperty( "C", "0-13-142644-3" );
       libros.setProperty( "C++", "0-13-185757-6" );
       libros.setProperty( "Visual Basic 2005", "0-13-186900-0" );
       libros.setProperty( "Visual C# 2005", "0-13-152523-9" );
    } // fin del constructor

    protected ApplicationBean1 getApplicationBean1()
    {
       return (ApplicationBean1) getBean( "ApplicationBean1" );
    } // fin del método getApplicationBean1

    protected RequestBean1 getRequestBean1()
    {
       return (RequestBean1) getBean( "RequestBean1" );
    } // fin del método getRequestBean1

    protected SessionBean1 getSessionBean1()
    {
       return (SessionBean1) getBean( "SessionBean1" );
    } // fin del método getSessionBean1

    public void init()
    {
       super.init();
       try
       {
          _init();
       } // fin de try
       catch ( Exception e )
       {
          log( "Error al inicializar Opciones", e );
          throw e instanceof FacesException ? (FacesException) e:
             new FacesException( e );
       } // fin de catch
    } // fin del método init

    public void preprocess()
    {
       // cuerpo vacío
    } // fin del método preprocess

    public void prerender()
    {
       // cuerpo vacío
    } // fin del método prerender

    public void destroy()
    {
       // cuerpo vacío
    } // fin del método destroy
```

Figura 26.30 | Bean de página que almacena las selecciones de lenguajes en una propiedad `SessionBean`. (Parte 2 de 3).

```java

   // manejador de acciones para el botón enviar, almacena los lenguajes seleccionados
   // en ámbito de sesión para obtenerlos al hacer recomendaciones de libros.
   public String enviar_action()
   {
      String msg = "Bienvenido a las sesiones!  Usted ";

      // si el usuario hizo una selección
      if ( getListaLenguajes().getSelected() != null )
      {
         String lenguaje = listaLenguajes.getSelected().toString();
         msg += "selecciono " + lenguaje + ".";

         // obtiene el número ISBN del libro para el lenguaje dado.
         String ISBN = libros.getProperty( lenguaje );

         // agrega la selección al objeto Properties de SessionBean1
         Properties selections = getSessionBean1().getLenguajesSeleccionados();
         Object resultado = selections.setProperty( lenguaje, ISBN );

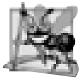
         // incrementa numSelecciones en el objeto SessionBean1 y actualiza
         // lenguajesSeleccionados si el usuario no ha realizado esta selección
         // antes
         if ( resultado == null )
         {
            int numSelec = getSessionBean1().getNumSelecciones();
            getSessionBean1().setNumSelecciones(++numSelec);
         } // fin de if
      } // fin de if
      else
         msg += "no selecciono un lenguaje.";

      etiquetaRespuesta.setValue( msg );
      listaLenguajes.setRendered( false );
      etiquetaLenguaje.setRendered( false );
      enviar.setRendered( false );
      etiquetaRespuesta.setRendered( true );
      etiquetaNumSelec.setRendered( true );
      numSelec.setRendered( true );
      vinculoLenguajes.setRendered( true );
      vinculoRecomendaciones.setRendered( true );
      return null;
   } // fin del método enviar_action

   // vuelve a mostrar los componentes usados para permitir al usuario
   // seleccionar un lenguaje.
   public String vinculoLenguajes1_action() {
      etiquetaRespuesta.setRendered( false );
      etiquetaNumSelec.setRendered( false );
      numSelec.setRendered( false );
      vinculoLenguajes.setRendered( false );
      vinculoRecomendaciones.setRendered( false );
      listaLenguajes.setRendered( true );
      etiquetaLenguaje.setRendered( true );
      enviar.setRendered( true );
      return null;
    } // fin del método vinculoLenguajes1_action
} // fin de la clase Opciones
```

Figura 26.30 | Bean de página que almacena las selecciones de lenguajes en una propiedad `SessionBean`. (Parte 3 de 3).

El manejador de acciones del componente `Botón` llamado `enviar` (líneas 280 a 319) almacena las selecciones del usuario en el objeto `SessionBean1` e incrementa el número de selecciones realizadas, si es necesario. En la línea 294 se obtiene el objeto `Properties` del objeto `SessionBean1` que contiene las selecciones del usuario. En la línea 295 se agrega la selección actual al objeto `Properties`. El método `setProperty` devuelve el valor previamente asociado con la nueva clave, o `null` si esta calve no se había almacenado ya en el objeto `Properties`. Si al agregar la nueva propiedad se devuelve `null`, entonces el usuario ha realizado una nueva selección. En este caso, en las líneas 302 y 303 se incrementa la propiedad `numSelecciones` en el objeto `SessionBean1`. En las líneas 309 a 317 y en el manejador de acciones `vinculoLenguajes1` (líneas 323 a 334) se controlan los componentes que se mostrarán en la página, igual que en los ejemplos de cookies.



Observación de ingeniería de software 26.2

Un beneficio de usar las propiedades de `SessionBean` *(en vez de cookies) es que pueden almacenar cualquier tipo de objeto (no sólo objetos* `String`*) como valores de atributos. Esto nos proporciona más flexibilidad para mantener la información de estado del cliente.*

Cómo mostrar las recomendaciones con base en los valores de sesiones

Al igual que en el ejemplo de las cookies, esta aplicación proporciona un vínculo a `Recomendaciones.jsp` (figura 26.31), el cual muestra una lista de recomendaciones de libros con base en los lenguajes seleccionados por el usuario. Es idéntico al archivo `Recomendaciones.jsp` del ejemplo de las cookies (figura 26.23).

```
<?xml version = "1.0" encoding = "UTF-8"?>

<!-- Fig. 26.31: Recomendaciones.jsp -->
<!-- Archivo JSP que muestra las recomendaciones de libros con base en los   -->
<!-- lenguajes seleccionados que se almacenan en el ámbito de sesión. -->
<jsp:root version = "1.2" xmlns:f = "http://java.sun.com/jsf/core"
   xmlns:h = "http://java.sun.com/jsf/html" xmlns:jsp =
   "http://java.sun.com/JSP/Page" xmlns:ui= "http://www.sun.com/web/ui">
   <jsp:directive.page contentType = "text/html; charset = UTF-8"
      pageEncoding = "UTF-8"/>
   <f:view>
      <ui:page binding = "#{Recomendaciones.page}" id = "page">
         <ui:html binding = "#{Recomendaciones.html}" id = "html">
            <ui:head binding = "#{Recomendaciones.head}" id = "head">
               <ui:link binding = "#{Recomendaciones.link}" id = "link"
                  url = "/resources/stylesheet.css"/>
               </ui:head>
               <ui:body binding = "#{Recomendaciones.body}" id = "body"
                 style = "-rave-layout: grid">
                 <ui:form binding = "#{Recomendaciones.form}" id = "form">
                    <ui:label binding = "#{Recomendaciones.etiquetaLenguaje}"
                       for = "cuadroListaLibros" id = "etiquetaLenguaje" style =
                       "font-size: 20px; font-weight: bold; left: 24px; top:
                       24px; position: absolute" text = "Recomendaciones"/>
                    <ui:listbox binding = "#{Recomendaciones.cuadroListaLibros}"
                       id = "cuadroListaLibros" items = "#{Recomendaciones.
                       cuadroListaLibrosDefaultOptions.options}" rows = "6"
                       style= "left: 24px; top: 72px; position: absolute;
                       width: 360px"/>
                    <ui:hyperlink action = "case1" binding =
                       "#{Recomendaciones.vinculoOpciones}" id = "vinculoOpciones"
                       style = "left: 24px; top: 192px; position: absolute"
                       text = "Haga clic aquí para seleccionar otro lenguaje."/>
```

Figura 26.31 | Archivo JSP que muestra las recomendaciones de libros con base en los lenguajes seleccionados que se almacenan en el ámbito de sesión. (Parte 1 de 2).

```
                </ui:form>
            </ui:body>
        </ui:html>
    </ui:page>
  </f:view>
</jsp:root>
```

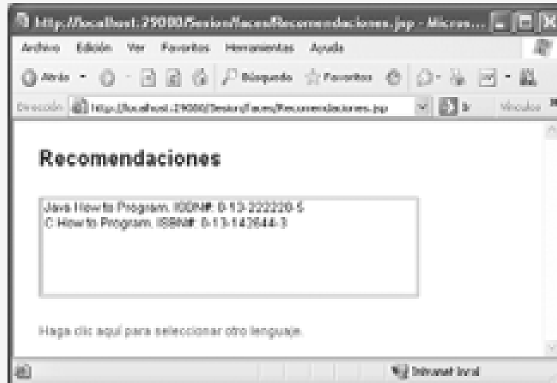


Figura 26.31 | Archivo JSP que muestra las recomendaciones de libros con base en los lenguajes seleccionados que se almacenan en el ámbito de sesión. (Parte 2 de 2).

Bean de página que crea recomendaciones de libros a partir de una propiedad de `SessionBean`

La figura 26.32 presenta el bean de página para `Recomendaciones.jsp`. De nuevo, gran parte de este código es similar al bean de página utilizado en el ejemplo de las cookies. Sólo hablaremos de las nuevas características.

```
// Fig. 26.32: Recomendaciones.java
// Bean de página que muestra las recomendaciones de libros con base en
// una propiedad de un objeto SessionBean.
package sesion;

import com.sun.rave.web.ui.appbase.AbstractPageBean;
import com.sun.rave.web.ui.component.Body;
import com.sun.rave.web.ui.component.Form;
import com.sun.rave.web.ui.component.Head;
import com.sun.rave.web.ui.component.Html;
import com.sun.rave.web.ui.component.Link;
import com.sun.rave.web.ui.component.Page;
import javax.faces.FacesException;
import com.sun.rave.web.ui.component.Listbox;
import com.sun.rave.web.ui.component.Label;
import com.sun.rave.web.ui.component.Hyperlink;
import com.sun.rave.web.ui.model.DefaultOptionsList;
import java.util.Enumeration;
import com.sun.rave.web.ui.model.Option;
import java.util.Properties;

public class Recomendaciones extends AbstractPageBean
{
   private int __placeholder;
```

Figura 26.32 | Bean de página que muestra las recomendaciones de libros con base en una propiedad de un objeto `SessionBean`. (Parte 1 de 3).

```
25
26    private void _init() throws Exception
27    {
28        // cuerpo vacío
29    } // fin del método _init
30
31    // para ahorrar espacio, omitimos el código de las líneas 31 a 150. El código
32    // fuente completo se proporciona con los ejemplos de este capítulo.
33
151   public Recomendaciones()
152   {
153      // constructor vacío
154   } // fin del constructor
155
156   protected RequestBean1 getRequestBean1()
157   {
158      return (RequestBean1) getBean( "RequestBean1" );
159   } // fin del método getRequestBean1
160
161   protected ApplicationBean1 getApplicationBean1()
162   {
163      return (ApplicationBean1) getBean( "ApplicationBean1" );
164   } // fin del método getApplicationBean1
165
166   protected SessionBean1 getSessionBean1()
167   {
168      return (SessionBean1) getBean( "SessionBean1" );
169   } // fin del método getSessionBean1
170
171   public void init()
172   {
173      super.init();
174      try
175      {
176         _init();
177      } // fin de try
178      catch ( Exception e )
179      {
180         log( "Error al inicializar Recomendaciones", e );
181         throw e instanceof FacesException ? (FacesException) e:
182            new FacesException( e );
183      } // fin de catch
184   } // fin del método init
185
186   public void preprocess()
187   {
188      // cuerpo vacío
189   } // fin del método preprocess
190
191   public void prerender()
192   {
193      // obtiene las selecciones del usuario y el número de selecciones realizadas
194      Properties lenguajes = getSessionBean1().getLenguajesSeleccionados();
195      Enumeration enumSelecciones = lenguajes.propertyNames();
196      int numSeleccionados = getSessionBean1().getNumSelecciones();
197
198      Option [] recomendaciones;
199
```

Figura 26.32 | Bean de página que muestra las recomendaciones de libros con base en una propiedad de un objeto SessionBean. (Parte 2 de 3).

```
200         // si por lo menos se hizo una selección
201         if ( numSeleccionados > 0 )
202         {
203            recomendaciones = new Option[ numSeleccionados ];
204
205            for( int i = 0; i < numSeleccionados; i++ )
206            {
207               String lenguaje = (String) enumSelecciones.nextElement() ;
208               recomendaciones[ i ] = new Option( lenguaje +
209                  " How to Program.       ISBN#: " +
210                  lenguajes.getProperty( lenguaje ) );
211            } // fin de for
212         } // fin de if
213         else
214         {
215            recomendaciones = new Option[ 1 ];
216            recomendaciones[ 0 ] = new Option( "No hay recomendaciones.
217               Seleccione un lenguaje." );
218         } // fin de else
219
220         cuadroListaLibros.setItems(recomendaciones);
221      } // fin del método prerender
222
223      public void destroy()
224      {
225         // cuerpo vacío
226      } // fin del método destroy
227   } // fin de la clase Recomendaciones
```

Figura 26.32 | Bean de página que muestra las recomendaciones de libros con base en una propiedad de un objeto `SessionBean`. (Parte 3 de 3).

En la línea 194 se obtiene el objeto `Properties` que contiene las selecciones que hizo el usuario del objeto `SessionBean1`, y en la línea 195 se obtiene una enumeración de todas las claves en ese objeto `Properties`. En la línea 196 se obtiene el número de selecciones que se realizaron a partir del objeto `SessionBean1`. Si se hizo alguna selección, en la línea 208 se construye un arreglo `Option` de tamaño apropiado para mostrar las selecciones en el elemento `ui:listBox` de `Recomendaciones.jsp`. En las líneas 205 a 211 se agrega cada una de las selecciones del usuario a este arreglo `Option`. En la línea 207 se obtiene la siguiente clave de la enumeración de teclas, y en las líneas 208 a 210 se agrega una recomendación al arreglo `Option`.

26.8 Conclusión

En este capítulo presentamos el desarrollo de aplicaciones Web mediante el uso de JavaServer Pages y JavaServer Faces en Java Studio Creator 2. Empezamos hablando sobre las transacciones HTTP simples que se llevan a cabo al solicitar y recibir una página Web a través de un navegador Web. Después hablamos sobre los tres niveles (es decir, el nivel cliente o superior, el nivel de lógica comercial o intermedio y el nivel de información o inferior) que conforman a la mayoría de las aplicaciones Web.

Aprendió acerca de la función que desempeñan los archivos JSP y los archivos de bean de página, y la relación entre ellos. Aprendió a utilizar Java Studio Creator 2 para compilar y ejecutar aplicaciones Web. También aprendió a crear aplicaciones Web en forma visual, mediante el uso de las herramientas de arrastrar y soltar de Java Studio Creator 2.

Demostramos varios componentes JSF comunes para mostrar texto e imágenes en las páginas Web. También hablamos sobre los componentes de validación y los métodos de validación personalizados, los cuales nos permiten asegurar que la entrada del usuario cumpla con ciertos requerimientos.

Hablamos sobre los beneficios de mantener la información del usuario a través de varias páginas de un sitio Web. Después demostramos cómo se pueden incluir dichas funcionalidades en una aplicación Web, mediante el

uso de cookies o propiedades en la clase `SessionBean`. En el siguiente capítulo continuaremos nuestra discusión acerca del desarrollo de aplicaciones Web. Aprenderá a utilizar una base de datos desde una aplicación Web, a utilizar varios de los componentes JSF habilitados para AJAX de la biblioteca Java Blueprints de Sun, y a utilizar los formularios virtuales.

26.9 Recursos Web

`developers.sun.com/prodtech/javatools/jscreator`
Presenta las generalidades acerca de Java Studio Creator 2 e incluye artículos, foros, demostraciones de productos y vínculos a recursos útiles, relevantes para la construcción de aplicaciones Web en Java Studio Creator 2.

`developers.sun.com/prodtech/javatools/jscreator/index.jsp`
El centro de Java Studio Creator de Sun, tiene todo lo que el programador necesita para empezar a trabajar. Descargue el IDE sin costo y dé un vistazo a la ficha de aprendizaje (Learning) para los tutoriales de Java Studio Creator.

`developers.sun.com/prodtech/javatools/jscreator/learning/tutorials/index.jsp`
Proporciona docenas de tutoriales, desde tips acerca de cómo empezar a trabajar con Java Studio Creator 2, hasta instrucciones de características específicas acerca de cómo utilizar muchas facetas del IDE.

`developers.sun.com/prodtech/javatools/jscreator/reference/docs/apis/`
La documentación para Java Studio Creator 2.

`java.sun.com/javaee/javaserverfaces/`
Este sitio oficial de Sun proporciona la documentación para JavaServer Faces, junto con vínculos hacia artículos y tutoriales relacionados.

`www.netbeans.org/products/visualweb/`
Aquí puede obtener el Netbeans Visual Web Pack 5.5 para Netbeans 5.5.

`java.sun.com/javaee/5/docs/tutorial/doc/bnaph.html`
El tutorial de JavaServer Faces de Java EE 5.

`jsftutorials.net/`
Vínculos a tutoriales y artículos generales sobre JavaServer Faces.

`javaserverfaces.dev.java.net`
Descargue la versión más reciente de la implementación de JavaServer Faces de Sun.

`java.sun.com/javaee/javaserverfaces/reference/api/`
Documentación sobre la biblioteca de etiquetas, la API y el Standard RenderKit para todas las versiones de JSF.

`java.sun.com/javaee/5/docs/tutorial/doc/JSFCustom.html`
Un tutorial acerca de cómo crear componentes JSF personalizados.

`bpcatalog.dev.java.net/nonav/webtier/index.html`
El catálogo de soluciones de Java BluePrints contiene ejemplos de código reutilizable para diseñar aplicaciones Web mediante el uso de JavaServer Faces y AJAX.

Resumen

Sección 26.1 Introducción

- Las aplicaciones basadas en Web crean contenido para los clientes navegadores Web.
- AJAX ayuda a las aplicaciones basadas en Web a proporcionar la interactividad y capacidad de respuesta que los usuarios esperan generalmente de las aplicaciones de escritorio.

Sección 26.2 Transacciones HTTP simples

- El Protocolo de transferencia de hipertexto (HTTP) especifica un conjunto de métodos y encabezados que permiten a los clientes y servidores interactuar e intercambiar información de una manera uniforme y confiable.
- En su forma más simple, una página Web no es nada más que un documento XHTML que contiene marcado para describir a un navegador Web cómo debe mostrar y dar formato a la información del documento.
- Los documentos XHTML pueden contener datos de hipertexto (hipervínculos) que vinculan a distintas páginas, o a otras partes de la misma página cuando el usuario hace clic en el vínculo.

- HTTP utiliza URIs (Identificadores uniformes de recursos) para identificar los datos en Internet.
- Los URIs que especifican las ubicaciones de los documentos se llaman URLs (Localizadores uniformes de recursos). Los URLs comunes hacen referencia a archivos o directorios, y pueden hacer referencia a objetos que realicen tareas complejas.
- Un URL contiene información que lleva a un navegador al recurso que el usuario desea utilizar. Las computadoras que ejecutan software de servidor Web hacen que dichos recursos estén disponibles.
- Cuando un navegador Web recibe un URL, realiza una transacción HTTP simple para obtener y mostrar la página Web que se encuentra en esa dirección.
- El método `GET` de HTTP indica que el cliente desea obtener un recurso del servidor.
- Los encabezados HTTP proporcionan información acerca de los datos que se envían a un servidor, como el tipo MIME.
- MIME (Extensiones de correo Internet multipropósito) es un estándar de Internet que especifica formatos de datos, para que los programas puedan interpretar los datos en forma correcta.

Sección 26.3 Arquitectura de aplicaciones multinivel

- Las aplicaciones basadas en Web son aplicaciones multinivel (o de *n* niveles), que dividen la funcionalidad en niveles separados que, por lo general, residen en computadoras separadas.
- El nivel inferior (también conocido como el nivel de datos o de información) mantiene los datos de la aplicación. Por lo general, este nivel almacena los datos en un sistema de administración de bases de datos relacionales (RDBMS).
- El nivel intermedio implementa la lógica de negocios, de controlador y de presentación para controlar las interacciones entre los clientes de la aplicación y sus datos. El nivel intermedio actúa como intermediario entre los datos en el nivel de información y los clientes de la aplicación.
- La lógica de control del nivel intermedio procesa las peticiones de los clientes y obtiene datos de la base de datos.
- La lógica de presentación del nivel intermedio procesa los datos del nivel de información y presenta el contenido al cliente.
- Por lo general, las aplicaciones Web presentan datos a los clientes en forma de documentos XHTML.
- La lógica comercial en el nivel intermedio hace valer las reglas comerciales y asegura que los datos sean confiables antes de que la aplicación servidor actualice la base de datos, o presente los datos a los usuarios.
- Las reglas comerciales dictan la forma en que los clientes pueden o no acceder a los datos de la aplicación, y la forma en que ésta procesa los datos.
- El nivel superior (nivel cliente) es la interfaz de usuario de la aplicación, la cual recopila los datos de entrada y de salida. Los usuarios interactúan en forma directa con la aplicación a través de la interfaz de usuario que, por lo general, es el navegador Web.
- En respuesta a las acciones del usuario, el nivel cliente interactúa con el nivel intermedio para hacer peticiones y obtener datos del nivel de información. Después, el nivel cliente muestra los datos obtenidos para el usuario. El nivel cliente nunca interactúa directamente con el nivel de información.

Sección 26.4 Tecnologías Web de Java

- Las tecnologías Web de Java evolucionan en forma continua, para ofrecer a los desarrolladores niveles mayores de abstracción, y una mayor separación de los niveles de la aplicación. Esta separación facilita el mantenimiento y la extensibilidad de las aplicaciones Web.
- Java Studio Creator 2 nos permite desarrollar la GUI de una aplicación Web mediante una herramienta de diseño tipo "arrastrar y soltar", mientras que podemos manejar la lógica comercial en clases de Java separadas.

Sección 26.4.1 Servlets

- Los servlets utilizan el modelo petición-respuesta HTTP de comunicación entre cliente y servidor.
- Los servlets extienden la funcionalidad de un servidor, al permitir que éste genere contenido dinámico. Un contenedor de servlets, ejecuta los servlets e interactúa con ellos.
- Los paquetes `javax.servlet` y `javax.servlet.http` contienen las clases e interfaces para definir servlets.
- El contenedor de servlets recibe peticiones HTTP de un cliente y dirige cada petición al servlet apropiado. El servlet procesa la petición y devuelve una respuesta apropiada al cliente; por lo general en forma de un documento XHTML o XML.
- Todos los servlets implementan a la interfaz `Servlet` del paquete `javax.servlet`, la cual asegura que cada servlet se pueda ejecutar en el marco de trabajo proporcionado por el contenedor de servlets. La interfaz `Servlet` declara métodos que el contenedor de servlets utiliza para administrar el ciclo de vida del servlet.
- El ciclo de vida de un servlet empieza cuando el contenedor de servlets lo carga en memoria; por lo general, en respuesta a la primera petición del servlet. El contenedor invoca al método `init` del servlet, el cual se llama sólo

una vez durante el ciclo de vida de un servlet para inicializarlo. Una vez que `init` termina su ejecución, el servlet está listo para responder a su primera petición. Todas las peticiones se manejan mediante el método `service` de un servlet, el cual recibe la petición, la procesa y envía una respuesta al cliente. Se hace una llamada al método `service` por cada petición. Cuando el contenedor de servlets termina el servlet, se hace una llamada al método `destroy` del servlet para liberar los recursos que éste ocupa.

Sección 26.4.2 JavaServer Pages

- La tecnología JavaServer Pages (JSP) es una extensión de la tecnología de los servlets. El contenedor de JSPs traduce cada JSP y la convierte en un servlet.
- A diferencia de los servlets, las JSPs nos ayudan a separar la presentación del contenido.
- Las JavaServer Pages permiten a los programadores de aplicaciones Web crear contenido dinámico mediante la reutilización de componentes predefinidos, y mediante la interacción con componentes que utilizan secuencias de comandos del lado servidor.
- Los programadores de JSPs pueden utilizar componentes especiales de software llamados JavaBeans, y bibliotecas de etiquetas personalizadas que encapsulan una funcionalidad dinámica y compleja.
- Las bibliotecas de etiquetas personalizadas permiten a los desarrolladores de Java ocultar el código para acceder a una base de datos y otras operaciones complejas mediante etiquetas personalizadas. Para usar dichas herramientas, sólo tenemos que agregar las etiquetas personalizadas a la página. Esta simpleza permite a los diseñadores de páginas Web, que no estén familiarizados con Java, mejorar las páginas Web con poderoso contenido dinámico y capacidades de procesamiento.
- Las clases e interfaces de JSP se encuentran en los paquetes `javax.servlet.jsp` y `javax.servlet.jsp.tagext`.
- Hay cuatro componentes clave para las JSPs: directivas, acciones, elementos de secuencia de comandos y bibliotecas de etiquetas.
- Las directivas son mensajes para el contenedor de JSPs que nos permiten especificar configuraciones de páginas, incluir contenido de otros recursos y especificar bibliotecas de etiquetas personalizadas para usarlas en las JSPs.
- Las acciones encapsulan la funcionalidad en etiquetas predefinidas que los programadores pueden incrustar en JSPs. A menudo, las acciones se realizan con base en la información que se envía al servidor como parte de una petición específica de un cliente. También pueden crear objetos de Java para usarlos en las JSPs.
- Los elementos de secuencia de comandos permiten al programador insertar código de Java que interactúe con los componentes en una JSP para realizar el procesamiento de peticiones.
- Las bibliotecas de etiquetas permiten a los programadores crear etiquetas personalizadas, y a los diseñadores de páginas Web manipular el contenido de las JSPs sin necesidad de tener un conocimiento previo sobre Java.
- La Biblioteca de etiquetas estándar de JavaServer Pages (JSTL) proporciona la funcionalidad para muchas tareas de aplicaciones Web comunes.
- Las JSPs pueden contener otro tipo de contenido estático, como marcado XHTML o XML, el cual se conoce como datos de plantilla fija o texto de plantilla fija. Cualquier texto literal en una JSP se traduce en una literal `String` en la representación de la JSP en forma de servlet.
- Cuando un servidor habilitado para JSP recibe la primera petición para una JSP, el contenedor de JSPs traduce esa JSP en un servlet, el cual maneja la petición actual y las futuras peticiones a esa JSP.
- Las JSPs se basan en el mismo mecanismo de petición-respuesta que los servlets para procesar las peticiones de los clientes, y enviar las respuestas.

Sección 26.4.3 JavaServer Faces

- JavaServer Faces (JSF) es un marco de trabajo para aplicaciones Web que simplifica el diseño de la interfaz de usuario de una aplicación, y separa aún más la presentación de una aplicación Web de su lógica comercial.
- Un marco de trabajo simplifica el desarrollo de aplicaciones, al proporcionar bibliotecas y (algunas veces) herramientas de software para ayudar al programador a organizar y crear sus aplicaciones.
- JSF proporciona bibliotecas de etiquetas personalizadas que contienen componentes de interfaz de usuario, los cuales simplifican el diseño de páginas Web. JSF también incluye APIs para manejar eventos de componentes.
- El programador diseña la apariencia visual de una página con JSF, agregando etiquetas a un archivo JSP y manipulando sus atributos. El programador define el comportamiento de la página por separado, en un archivo de código fuente de Java relacionado.

Sección 26.4.4 Tecnologías Web en Java Studio Creator 2

- Las aplicaciones Web de Java Studio Creator 2 consisten en una o más páginas Web JSP, integradas en el marco de trabajo JavaServer Faces. Cada archivo JSP tiene la extensión `.jsp` y contiene los elementos de la GUI de la página Web.

- Java Studio Creator 2 nos permite diseñar páginas en forma visual, al arrastrar y soltar componentes JSF en una página; también podemos personalizar una página Web al editar su archivo `.jsp` en forma manual.
- Cada archivo JSP que se crea en Java Studio Creator 2 representa una página Web, y tiene su correspondiente clase JavaBean, denominada bean de página.
- Una clase JavaBean debe tener un constructor predeterminado (o sin argumentos), junto con métodos *obtener (get)* y *establecer (set)* para todas sus propiedades.
- El bean de página define las propiedades para cada uno de los elementos de la página, y contiene manejadores de eventos, métodos de ciclo de vida de las páginas y demás código de soporte para la aplicación Web.
- Toda aplicación Web creada con Java Studio Creator 2 tiene un bean de página, un objeto `RequestBean`, un objeto `SessionBean` y un objeto `ApplicationBean`.
- El objeto `RequestBean` se mantiene en `ámbito de petición`; este objeto existe sólo mientras dure una petición HTTP.
- Un objeto `SessionBean` tiene `ámbito de sesión`; el objeto existe durante una sesión de navegación del usuario, o hasta que se agota el tiempo de la sesión. Hay un único objeto `SessionBean` para cada usuario.
- El objeto `ApplicationBean` tiene `ámbito de aplicación`; este objeto es compartido por todas las instancias de una aplicación y existe mientras que la aplicación esté desplegada en un servidor Web. Este objeto se utiliza para almacenar datos a nivel de aplicación o para procesamiento; sólo existe una instancia para la aplicación, sin importar el número de sesiones abiertas.

Sección 26.5.1 Análisis de un archivo JSP

- Java Studio Creator 2 genera un archivo JSP en respuesta a las interacciones del programador con el Editor visual.
- Todas las JSPs tienen un elemento `jsp:root` con un atributo `version` para indicar la versión de JSP que se utiliza, y uno o más atributos `xmlns`. Cada atributo `xmlns` especifica un prefijo y un URL para una biblioteca de etiquetas, lo cual permite a la página usar las etiquetas especificadas en esa biblioteca.
- Todas las JSPs generadas por Java Studio Creator 2 incluyen las bibliotecas de etiquetas para la biblioteca de componentes JSF básicos, la biblioteca de componentes JSF de HTML, la biblioteca de componentes JSP estándar y la biblioteca de componentes JSP de interfaz de usuario.
- El atributo `contentType` del elemento `jsp:directive.page` especifica el tipo MIME y el conjunto de caracteres utilizado por la página. El atributo `pageEncoding` especifica la codificación de caracteres utilizada por la página de origen. Estos atributos ayudan al cliente a determinar cómo desplegar el contenido.
- Todas las páginas que contienen componentes JSF se representan en un árbol de componentes, con el elemento JSF raíz `f:view` (de tipo `UIViewRoot`). Todos los elementos de los componentes JSF se colocan en este elemento.
- Muchos elementos de página `ui` tienen un atributo `binding` para enlazar sus valores a las propiedades en los JavaBeans de la aplicación Web. Para realizar estos enlaces, se utiliza el Lenguaje de expresiones de JSF.
- El elemento `ui:head` tiene un atributo `title` que especifica el título de la página.
- Un elemento `ui:link` se puede utilizar para especificar la hoja de estilos CSS utilizada por una página.
- Un elemento `ui:body` define el cuerpo de la página.
- Un elemento `ui:form` define a un formulario en una página.
- Un componente `ui:staticText` muestra texto que no cambia.
- Los elementos JSP se asignan a elementos XHTML para desplegarlos en un navegador. Los mismos elementos JSP se pueden asignar a distintos elementos XHTML, dependiendo del navegador cliente y de las configuraciones de las propiedades de los componentes.
- Por lo general, un componente `ui:staticText` se asigna a un elemento `span` de XHTML. Un elemento `span` contiene texto que se muestra en una página Web, y se utiliza para controlar el formato del texto. El atributo `style` de un elemento `ui:staticText` se representará como parte del atributo `style` del elemento `span` correspondiente cuando el navegador despliegue la página.

Sección 26.5.2 Análisis de un archivo de bean de página

- Las clases de bean de página heredan de la clase `AbstractPageBean` (paquete `com.sun.rave.web.ui.appbase`), la cual proporciona métodos para el ciclo de vida de la página.
- El paquete `com.sun.rave.web.ui.component` incluye clases para muchos componentes JSF básicos.
- Un componente `ui:staticText` es un objeto `StaticText` (paquete `com.sun.rave.web.ui.component`).

Sección 26.5.3 Ciclo de vida del procesamiento de eventos

- El modelo de aplicación de Java Studio Creator 2 coloca varios métodos (`init`, `preprocess`, `prerender` y `destroy`) en el bean de página que se enlaza con el ciclo de vida de procesamiento de eventos JSF. Estos métodos representan cuatro etapas principales: inicialización, pre-procesamiento, pre-despliegue y destrucción.

- El contenedor de JSPs llama al método `init` la primera vez que se solicita la página, y en las peticiones de devolución de envío (postback). Una petición de devolución de envío ocurre cuando se envían formularios, y la página (junto con su contenido) se envía al servidor para su procesamiento.
- El método `init` invoca a la versión de su superclase, y después trata de llamar al método `_init`, el cual maneja las tareas de inicialización de los componentes.
- El método `preprocess` se llama después de `init`, pero sólo si la página está procesando una petición de devolución de envío. El método `prerender` se llama justo antes de que el navegador despliegue una página. Este método se debe utilizar para establecer las propiedades de los componentes; las propiedades que se establezcan antes de tiempo (como en el método `init`) pueden sobrescribirse antes de que el navegador llegue a desplegar la página.
- El método `destroy` se llama después de haber desplegado la página, pero sólo si se llamó al método `init`. Este método maneja las tareas tales como liberar los recursos utilizados para desplegar la página.

Sección 26.5.4 Relación entre la JSP y los archivos de bean de página

- El bean de página tiene una propiedad para cada elemento que aparece en el archivo JSP.

Sección 26.5.5 Análisis del XHTML generado por una aplicación Web de Java

- Para crear una nueva aplicación Web, seleccione **Archivo > Nuevo proyecto…** para mostrar el cuadro de diálogo **Nuevo proyecto**. En este cuadro de diálogo, seleccione **Web** en el panel **Categorías**, **Aplicación web JSF** en el panel **Proyectos** y haga clic en **Siguiente**. Especifique el nombre del proyecto y la ubicación. Haga clic en **Terminar** para crear el proyecto de aplicación Web.
- Al crear un nuevo proyecto, Java Studio Creator 2 crea una sola página Web llamada `Page1`. Esta página se abre de manera predeterminada en el Editor visual en modo **Diseño**, cuando el proyecto se carga por primera vez. A medida que el programador arrastra y suelta nuevos componentes en la página, el modo **Diseño** le permite ver cómo se desplegará la página en el navegador. El archivo JSP para esta página, llamado `Page1.jsp`, se puede ver haciendo clic en el botón **JSP** en la parte superior del Editor visual, o haciendo clic con el botón derecho del ratón en cualquier parte del Editor visual y seleccionando la opción **Editar origen JSP**.
- Para abrir el archivo de bean de página correspondiente, haga clic en el botón **Java** en la parte superior del Editor visual, o haga clic con el botón derecho del ratón en cualquier parte del Editor visual y seleccione **Editar origen Java Page1**.
- El botón **Previsualizar en navegador** en la parte superior de la ventana Editor visual le permite ver sus páginas en un navegador sin tener que crear y ejecutar la aplicación.
- El botón **Actualizar** vuelve a dibujar la página en el Editor visual.
- La lista desplegable **Tamaño de navegador de destino** nos permite especificar la resolución óptima del navegador para ver la página, y nos permite ver cuál será la apariencia de la página en distintas resoluciones de pantalla.
- La ventana **Proyectos** en la esquina inferior derecha del IDE muestra la jerarquía de todos los archivos del proyecto. El nodo **Páginas Web** contiene los archivos JSP e incluye la carpeta **resources**, la cual contiene la hoja de estilo CSS para el proyecto, y cualquier otro archivo que puedan necesitar las páginas para mostrarse en forma apropiada (como los archivos de imagen). El código fuente de Java, incluyendo el archivo de bean de página para cada página Web y los beans de aplicación, sesión y petición, se pueden encontrar bajo el nodo **Paquetes de origen**.
- El archivo **Navegación de página** define reglas para navegar por las páginas del proyecto, con base en el resultado de algún evento iniciado por el usuario, como hacer clic en un botón o en un vínculo. También podemos acceder al archivo **Navegación de página** si hacemos clic con el botón derecho del ratón en el Editor visual, estando en modo **Diseño**; para ello, seleccionamos la opción **Navegación de página…**.
- Los métodos `init`, `preprocess`, `prerender` y `destroy` se sobrescriben en cada bean de página. Aparte del método `init`, estos métodos están vacíos al principio. Sirven como receptáculos para que usted pueda personalizar el comportamiento de su aplicación Web.
- Por lo general, es conveniente cambiar el nombre de los archivos JSP y Java en el proyecto, de manera que los nombres sean relevantes para nuestra aplicación. Para ello, haga clic en el archivo JSP en la **Ventana Proyectos** y seleccione **Cambiar nombre** para mostrar el cuadro de diálogo **Cambiar nombre**. Escriba el nuevo nombre para el archivo. Si está activada la opción **Previsualizar todos los cambios**, aparecerá la **Ventana Refactorización** en la parte inferior del IDE cuando haga clic en **Siguiente >**. No se realizarán cambios hasta que haga clic en el botón **Refactorizar** de la **Ventana Refactorización**. Si no previsualiza los cambios, la refactorización se llevará a cabo cuando haga clic en el botón **Siguiente >** del cuadro de diálogo **Cambiar nombre**.
- La refactorización es el proceso de modificar el código fuente para mejorar su legibilidad y reutilización, sin cambiar su comportamiento; por ejemplo, al cambiar el nombre a los métodos o variables, o al dividir métodos extensos en métodos más cortos. Java Studio Creator 2 tiene herramientas de refactorización integradas, que automatizan ciertas tareas de refactorización.

- Para agregar un título, abra la página JSP en modo **Diseño**. En la ventana **Propiedades**, escriba el nuevo título enseguida de la propiedad **Título** y oprima *Intro*.
- Para agregar componentes a una página, arrástrelos y suéltelos desde la **Paleta** hacia la página en modo **Diseño**. Cada componente es un objeto que tiene propiedades, métodos y eventos. Puede establecer estas propiedades y eventos en la ventana **Propiedades**, o mediante programación en el archivo de bean de página. Los métodos *obtener (get)* y *establecer (set)* se agregan al archivo de bean de página para cada componente que agregamos a la página.
- Los componentes se despliegan usando el posicionamiento absoluto, de manera que aparezcan exactamente en donde se sueltan en la página.
- Java Studio Creator 2 es un editor WYSIWYG (What You See Is What You Get —Lo que ve es lo que obtiene); cada vez que realice un cambio a una página Web en modo **Diseño**, el IDE creará el marcado (visible en modo **JSP**) necesario para lograr los efectos visuales deseados que aparecen en modo **Diseño**.
- Después de diseñar la interfaz, puede modificar el bean de página para agregar su lógica comercial.
- La ventana **Esquema** muestra el bean de página y los beans de ámbito de petición, sesión y aplicación. Al hacer clic en un elemento en el árbol de componentes del bean de página, se selecciona el elemento en el Editor visual.
- Seleccione **Generar > Generar proyecto principal**, y después **Ejecutar > Ejecutar proyecto principal** para ejecutar la aplicación.
- Para agregar un proyecto que ya haya sido creado, oprima el ícono **Ejecutar proyecto principal** ( ) en la barra de herramientas que se encuentra en la parte superior del IDE.
- Si se hacen cambios a un proyecto, éste debe volver a generarse para que puedan reflejarse los cambios cuando se vea la aplicación en un navegador Web.
- Oprima *F5* para generar la aplicación y después ejecutarla en modo de depuración. Si escribe *<Ctrl> F5*, el programa se ejecutará sin la depuración habilitada.

Sección 26.5.6 Creación de una aplicación Web en Java Studio Creator 2

- El componente **Panel de cuadrícula** permite al diseñador especificar el número de columnas que debe contener la cuadrícula. Después se pueden soltar los componentes en cualquier parte dentro del panel, y éstos se reposicionarán automáticamente en columnas espaciadas de manera uniforme, en el orden en el que se suelten. Cuando el número de componentes excede al número de columnas, el panel desplaza los componentes adicionales hacia una nueva fila.
- Un componente **Imagen** (de la clase `Image`) inserta una imagen en una página Web. Las imágenes que se van a mostrar en una página Web se deben colocar en la carpeta `resources` del proyecto. Para agregar imágenes al proyecto, suelte un componente **Imagen** en la página y haga clic en el botón de elipsis que está a un lado de la propiedad **url** en la ventana **Propiedades**. A continuación se abrirá un cuadro de diálogo, en el que puede seleccionar la imagen a mostrar.
- Los componentes **Campo de texto** nos permiten obtener entrada de texto del usuario.
- Observe que el orden en el que se arrastran los componentes a la página es importante, ya que sus etiquetas JSP se agregan al archivo JSP en ese orden. El uso de tabuladores entre los componentes permite navegar por los componentes en el orden en el que se hayan agregado las etiquetas JSP al archivo JSP. Si desea que el usuario navegue por los componentes en cierto orden, debe arrastrarlos a la página en ese orden. De manera alternativa, puede establecer la propiedad **Tab Index** de cada campo de entrada en la ventana **Propiedades**. Un componente con un índice de tabulación de 1 será el primero en la secuencia de tabulaciones.
- Una **Lista desplegable** muestra una lista, de la cual el usuario puede seleccionar un elemento. Este objeto se puede configurar haciendo clic con el botón derecho en la lista desplegable en modo **Diseño** y seleccionando **Configurar opciones predeterminadas**, con lo cual se abrirá el cuadro de diálogo **Personalizador de opciones** para agregar opciones a la lista.
- Un componente **Hipervínculo** de la clase `Hyperlink` agrega un vínculo a una página Web. La propiedad `url` de este componente especifica el recurso que se solicita cuando un usuario hace clic en el hipervínculo.
- Un componente **Grupo de botones de selección** de la clase `RadioButtonGroup` proporciona una serie de botones de opción, de los cuales el usuario puede seleccionar sólo uno. Para editar las opciones, haga clic con el botón derecho del ratón en el componente y seleccione **Configurar opciones predeterminadas**.
- Un **Botón** es un componente JSF de la clase `Button` que desencadena una acción cuando es oprimido. Por lo general, un componente **Button** se asigna a un elemento `input` de XHTML con el atributo `type` establecido en `submit`.

Sección 26.6.2 Validación mediante los componentes de validación y validadores personalizados

- La validación ayuda a prevenir los errores de procesamiento, debido a una entrada del usuario incompleta o con un formato inapropiado.
- Un **Validador de longitud** determina si un campo contiene un número aceptable de caracteres.

- El **Validador de intervalo doble** y el **Validador de intervalo largo** determinan si la entrada numérica se encuentra dentro de intervalos aceptables.
- El paquete `javax.faces.validators` contiene las clases para estos validadores.
- Los componentes **Etiqueta** describen a otros componentes, y se pueden asociar con los campos de entrada del usuario si se establece su propiedad `for`.
- Los componentes **Mensaje** muestran mensajes de error cuando falla la validación.
- Para asociar un componente **Etiqueta** o **Mensaje** con otro componente, mantenga oprimidas las teclas *Ctrl* y *Mayús*, y después arrastre la etiqueta o mensaje hacia el componente apropiado.
- Establezca la propiedad `required` de un componente para asegurar que el usuario escriba datos para éste.
- Si agrega un componente de validación o un método de validación personalizado a un campo de entrada, la propiedad `required` de ese campo se debe establecer en `true` para que ocurra la validación.
- En el Editor visual, la etiqueta para un campo requerido se marca automáticamente con un asterisco color rojo.
- Si un usuario envía un formulario con un campo de texto vacío para el cual se requiere un valor, se mostrará el mensaje de error predeterminado para ese campo en su componente `ui:message` asociado.
- Para editar las propiedades de un **Validador de intervalo doble** o de un **Validador de intervalo largo**, haga clic en su nodo en la ventana **Esquema** en modo **Diseño** y establezca las propiedades `minimum` y `maximum` en la ventana **Propiedades**.
- Es posible limitar la longitud de la entrada del usuario sin validación, para lo cual hay que establecer la propiedad `maxLength` de un componente **Campo de texto**.
- Comparar la entrada del usuario con una expresión regular es una forma efectiva de asegurar que la entrada tenga un formato apropiado.
- Java Studio Creator 2 no proporciona componentes para validación mediante el uso de expresiones regulares, pero podemos agregar nuestros propios métodos de validación al archivo de bean de página.
- Para agregar un método de validación personalizado a un componente de entrada, haga clic con el botón derecho en el componente y seleccione **Editar manejador de eventos > validate** para crear un método de validación para el componente en el archivo de bean de página.

Sección 26.7 Rastreo de sesiones

- La personalización hace posible que los comercios electrónicos se comuniquen con eficiencia con sus clientes, y también mejora la habilidad del usuario para localizar los productos y servicios deseados.
- Existe una concesión entre el servicio de comercio electrónico personalizado y la protección de la privacidad. Algunos consumidores adoptan la idea del contenido personalizado, pero otros temen a las posibles consecuencias adversas, si la información que proporcionan a los comercios electrónicos es liberada o recolectada por tecnologías de rastreo.
- Para proporcionar servicios personalizados a los consumidores, los comercios electrónicos deben tener la capacidad de reconocer a los clientes cuando solicitan información de un sitio. Por desgracia, HTTP es un protocolo sin estado; no soporta conexiones persistentes que permitan a los servidores Web mantener información de estado, en relación con clientes específicos. Por lo tanto, los servidores Web no pueden determinar si una petición proviene de un cliente específico, o si una serie de peticiones provienen de uno o varios clientes.
- Para ayudar al servidor a diferenciar un cliente de otro, cada cliente debe identificarse a sí mismo con el servidor. El rastreo de clientes individuales, conocido como rastreo de sesiones, puede lograrse de varias formas. Una técnica popular utiliza cookies; otra utiliza el objeto `SessionBean`.
- Con los elementos `"hidden"`, un formulario Web puede escribir los datos de rastreo de sesión en un componente `form` en la página Web que devuelve al cliente, en respuesta a una petición previa. Cuando el usuario envía el formulario en la nueva página Web, todos los datos del formulario (incluyendo los campos `"hidden"`) se envían al manejador del formulario en el servidor Web. Con la reescritura de URLs, el servidor Web incrusta la información de rastreo de sesión directamente en los URLs de los hipervínculos en los que el usuario hace clic para enviar las subsiguientes peticiones al servidor Web.

Sección 26.7.1 Cookies

- Una cookie es una pieza de datos que, por lo general, se almacena en un archivo de texto en la computadora del usuario. Una cookie mantiene información acerca del cliente, durante y entre las sesiones del navegador.
- La primera vez que un usuario visita el sitio Web, su computadora podría recibir una cookie; después, esta cookie se reactiva cada vez que el usuario vuelve a visitar ese sitio. La información recolectada tiene el propósito de ser un registro anónimo que contiene datos, los cuales se utilizan para personalizar las visitas futuras del usuario al sitio Web.
- Cada interacción basada en HTTP entre un cliente y un servidor incluye un encabezado, el cual contiene información sobre la petición (cuando la comunicación es del cliente al servidor) o sobre la respuesta (cuando la comunicación es del servidor al cliente).

- Cuando una página recibe una petición, el encabezado incluye información como el tipo de petición y cualquier cookie que se haya enviado anteriormente del servidor, para almacenarse en el equipo cliente. Cuando el servidor formula su respuesta, la información del encabezado contiene cualquier cookie que el servidor desee almacenar en la computadora cliente, junto con más información, como el tipo MIME de la respuesta.
- La fecha de expiración de una cookie determina la forma en que ésta permanecerá en la computadora del cliente. Si no establecemos una fecha de expiración para la cookie, el navegador Web mantendrá la cookie mientras dure la sesión de navegación. En caso contrario, el navegador Web mantendrá la cookie hasta que llegue la fecha de expiración.
- Al establecer el manejador de acciones para un **Hipervínculo** podemos responder a un clic sin necesidad de redirigir al usuario hacia otra página.
- Para agregar un manejador de acciones a un **Hipervínculo** que también debe dirigir al usuario a otra página, debemos agregar una regla al archivo **Navegación de página**. Para editar este archivo, haga clic con el botón derecho del ratón en cualquier parte del Diseñador visual y seleccione **Navegación de página...**; después arrastre el **Hipervínculo** apropiado a la página de destino.
- Un objeto cookie es una instancia de la clase `Cookie` en el paquete `javax.servlet.http`.
- Un objeto de la clase `HttpServletResponse` (del paquete `javax.servlet.http`) representa la respuesta. Para acceder a este objeto, hay que invocar el método `getExternalContext` en el bean de página, y después invocar a `getResponse` en el objeto resultante.
- Un objeto de la clase `HttpServletRequest` (del paquete `javax.servlet.http`) representa la petición. Para obtener este objeto, hay que invocar al método `getExternalContext` en el bean de página, y después invocar a `getRequest` en el objeto resultante.
- El método `getCookies` de `HttpServletRequest` devuelve un arreglo de las cookies que se escribieron previamente en el cliente.
- Un servidor Web no puede acceder a las cookies creadas por los servidores en otros dominios.

Sección 26.7.2 Rastreo de sesiones con el objeto `SessionBean`

- Podemos llevar a cabo el rastreo de sesiones con la clase `SessionBean` que se proporciona en cada aplicación Web creada con Java Studio Creator 2. Cuando un nuevo cliente solicita una página Web en el proyecto, se crea un objeto `SessionBean`.
- Podemos acceder al objeto `SessionBean` a través de una sesión, invocando al método `getSessionBean` en el bean de página. Podemos usar el objeto `SessionBean` para acceder a las propiedades de sesión almacenadas.
- Para almacenar información en el objeto `SessionBean`, agregue propiedades a la clase `SessionBean`. Para agregar una propiedad, haga clic con el botón derecho del ratón en el nodo `SessionBean` en la ventana **Esquema** y seleccione **Agregar | Propiedad** para mostrar el cuadro de diálogo **Nuevo patrón de propiedad**. Configure la propiedad y haga clic en **Aceptar** para crearla.

Terminología

`AbstractPageBean`
acción en una JSP
`action`, atributo del elemento `form` de XHTML
aplicación basada en Web de tres niveles
aplicación de *n* niveles
aplicación multinivel
`ApplicationBean`
árbol de componentes
bean de página
biblioteca de etiquetas
bibilioteca de etiquetas personalizadas
búsqueda DNS
Button, componente JSF
Campo de texto, componente JSF
ciclo de vida del procesamiento de eventos
`com.sun.rave.web.ui.component`
contenedor de JSPs
contenedor de servlets
cookie
`Cuadro de lista`, componente JSF
datos de plantilla fija
desarrollo de aplicación Web
desplegar XHTML en un navegador Web
`destroy`, método del ciclo de vida de procesamiento de eventos
dirección IP
directiva en una JSP
directorio virtual
Diseño, modo
Editor visual
editor WYSIWYG (Lo que ve es lo que obtiene)
elemento de secuencia en una JSP
encabezado HTTP
entrada oculta en un formulario de XHTML
`escaped`, propiedad
`Esquema`, ventana
`Etiqueta`, componente JSF
etiqueta de XHTML

etiqueta final
etiqueta inicial
etiqueta personalizada
fecha de expiración de una cookie
`GET`, petición http
Grupo de botones de selección, componente JSF
hipertexto
hipervínculo
Hiperlink, componente JSF
host
Image, componente JSF
`init`, método del ciclo de vida de procesamiento de eventos
Java BluePrints
Java Studio Creator 2
JavaBeans
`javax.servlet`, paquete
`javax.servlet.http`, paquete
JSF (JavaServer Faces)
JSF, componentes
JSF, Lenguaje de expresiones
JSP (JavaServer Pages)
`.jsp`, extensión de archivo
JSTL (Biblioteca de etiquetas estándar de JSP)
Lista desplegable, componente JSF
`localhost`
lógica comercial
lógica de control
lógica de presentación
marcado de XHTML
marco de trabajo
mecanismo de extensión de etiquetas
Mensaje, componente JSF
`method`, atributo del elemento `form` de XHTML
método HTTP
MIME (Extensiones de correo Internet multipropósito)
nivel cliente
nivel de datos
nivel de información
nivel en una aplicación multinivel
nivel inferior
nivel intermedio
nivel superior
nombre de host
Paleta
Panel de cuadrícula, componente JSF
personalización
petición de devolución de envío (postback)
posicionamiento absoluto
`preprocess`, método del ciclo de vida de procesamiento de eventos
`prerender`, método del ciclo de vida del procesamiento de eventos
rastreo de sesiones
refactorización
regla comercial
`rendered`, propiedad
`RequestBean`
`required`, propiedad
service, método de la interfaz `Servlet`
servidor DNS (sistema de nombres de dominio)
servidor Web
servlet
`Servlet`, interfaz
`SessionBean`
`span`, elemento
Sun Application Server 8
texto de plantilla fija
Texto estático, componente JSF
`title`, elemento de XHTML
validación
Validador de intervalo doble, componente JSF
Validador de intervalo largo, componente JSF
Validador de longitud, componente JSF
XML
xmlns, atributos

Ejercicios de autoevaluación

26.1 Conteste con *verdadero* o *falso* a cada una de las siguientes proposiciones; en caso de ser *falso*, explique por qué.

a) Toda página Web JSP creada en Java Studio Creator 2 tiene sus propios archivos `ApplicationBean`, `SessionBean` y `RequestBean`.
b) El método `init` del ciclo de vida de procesamiento de eventos se invoca cada vez que se carga una página.
c) Cada componente en una página Web JSP está enlazado a una propiedad en el archivo Java de bean de página.
d) Un solo componente JSF puede tener varios componentes de validación colocados sobre él.
e) Si no se establece una fecha de expiración para una cookie, esa cookie se destruirá al final de la sesión del navegador.
f) Cada componente JSF se asigna sólo a un elemento XHTML correspondiente.
g) Las expresiones en la sintaxis del Lenguaje de expresiones JSF se delimitan por los signos `<!--` y `-->`.
h) El objeto `SessionBean` puede almacenar sólo propiedades primitivas y propiedades de tipo `String`.

26.2 Complete las siguientes oraciones:

a) Las aplicaciones Web contienen tres niveles básicos: ________, ________ y ________.

b) El componente JSF ________ se utiliza para mostrar mensajes de error, en caso de que falle la validación.

c) Un componente que comprueba la entrada en otro componente antes de enviar esa entrada al servidor se llama ________.

d) Cada clase de bean de página hereda de la clase ________.

e) Cuando una página se carga la primera vez, el evento ________ ocurre primero, seguido del evento ________.

f) El archivo ________ contiene la funcionalidad para una JSP.

g) Un ________ se puede utilizar en un método de validación personalizado para validar el formato de la entrada del usuario.

h) El arreglo de objetos `Cookie` almacenados en el cliente se puede obtener llamando al método `getCookies` en el objeto ________.

i) En una aplicación multinivel, el nivel ________ controla las interacciones entre los clientes de la aplicación y los datos de la misma.

Respuestas a los ejercicios de autoevaluación

26.1 a) Falso. Si una aplicación contiene varias JSPs, esas JSPs compartirán los beans de datos con ámbito. b) Falso. `init` se invoca la primera vez que se solicita la página, pero no en las actualizaciones de páginas. c) Verdadero. d) Verdadero. e) Verdadero. f) Falso. Un componente Web se puede asignar a un grupo de elementos de XHTML; las JSPs pueden generar marcado de XHTML complejo, a partir de componentes simples. g) Falso. `#{` y `}` delimitan las instrucciones del Lenguaje de expresiones JSF. h) Falso. Los beans de datos con ámbito pueden almacenar cualquier tipo de propiedad.

26.2 a) inferior (información), intermedio (lógica comercial), superior (cliente). b) **Mensaje**. c) validador. d) `AbstractPageBean`. e) `init`, `prerender`. f) bean de página. g) expresión regular. h) Petición (`HttpServletRequest`). i) intermedio.

Ejercicios

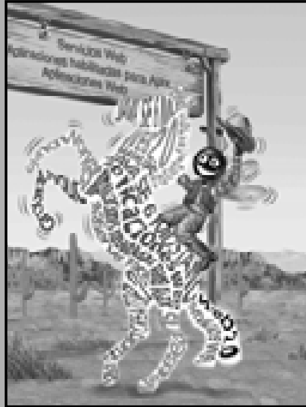
26.3 *(Modificación de `HoraWeb`)* Modifique el ejemplo `HoraWeb` para que contenga listas desplegables que permitan al usuario modificar las propiedades de los componentes **Texto estático** tales como `background-color`, `color` y `font-size`. Cuando se vuelva a cargar la página, deberá reflejar los cambios especificados en las propiedades del componente **Texto estático** que muestra la hora.

26.4 *(Modificación del formulario de registro)* Modifique la aplicación `ComponentesWeb` para agregar funcionalidad al botón **Registro**. Cuando el usuario haga clic en **Enviar**, valide todos los campos de entrada para asegurar que el usuario haya llenado el formulario por completo y haya introducido una dirección de e-mail válida, junto con un número telefónico válido. Después, lleve al usuario a otra página que muestre un mensaje indicando un registro exitoso, y que vuelva a imprimir la información de registro para el usuario.

26.5 *(Contador de visitas a las páginas con Cookies)* Cree un archivo JSP que utilice una cookie persistente (es decir, una cookie con una fecha de expiración en el futuro) para llevar la cuenta de cuántas veces el equipo cliente ha visitado la página. Use el método `setMaxAge` para hacer que la cookie permanezca en el equipo cliente durante un mes. Muestre el número de visitas a la página (es decir, el valor de la cookie) cada vez que se cargue la página.

26.6 *(Contador de visitas de páginas con **ApplicationBean**)* Cree un archivo JSP que utilice el objeto `ApplicationBean` para llevar la cuenta de cuántas veces se ha visitado una página. [*Nota:* si desplegara esta página en la Web, contaría el número de veces que un equipo haya solicitado la página, a diferencia del ejercicio anterior]. Muestre el número de visitas a la página (es decir, el valor de una propiedad `int` en el objeto `ApplicationBean`) cada vez que se cargue la página.

27



Lo que de cualquier forma sea bello, tiene su fuente de belleza en sí mismo, y es completo por sí solo; el elogio no forma parte de éste.

—Marcus Aurelius Antoninus

Hay algo en un rostro, un aire, y una gracia peculiar, que los pintores más audaces no pueden trazar.

—William Somerville

Cato dijo que la mejor forma de mantener los buenos actos en la memoria era refrescarlos con nuevos actos.

—Francis Bacon

Nunca olvido un rostro, pero en su caso haré una excepción.

—Groucho Marx

La pintura es sólo un puente que enlaza la mente del pintor con la del observador.

—Eugéne Delacroix

Aplicaciones Web: parte 2

OBJETIVOS

En este capítulo aprenderá a:

- Utilizar los proveedores de datos para acceder a las bases de datos desde las aplicaciones Web integradas en Java Studio Creator 2.
- Conocer los principios básicos y ventajas de la tecnología Ajax.
- Incluir componentes JSF habilitados para Ajax en un proyecto de aplicación Web de Java Studio Creator 2.
- Configurar formas virtuales que permiten a los subconjuntos de los componentes de entrada de un formulario ser enviados al servidor.

Plan general

27.1 Introducción

En este capítulo continuaremos nuestra discusión acerca del desarrollo de aplicaciones Web con varios conceptos avanzados. Hablaremos sobre cómo acceder, actualizar y realizar búsquedas en bases de datos en una aplicación Web, cómo agregar formularios virtuales a páginas Web para permitir que se envíen subconjuntos de los componentes de entrada de un formulario al servidor, y cómo usar las bibliotecas de componentes habilitados para Ajax, para mejorar el rendimiento de la aplicación y la capacidad de respuesta de los componentes.

Presentaremos una aplicación de libreta de direcciones desarrollada en tres etapas, para ilustrar estos conceptos. La aplicación está respaldada por una base de datos Java DB para almacenar los nombres de los contactos y sus direcciones.

La aplicación de libreta de direcciones presenta un formulario que permite al usuario introducir un nuevo nombre y dirección para almacenarlos en la libreta de direcciones, y muestra el contenido de esta libreta en formato tabular. También proporciona un formulario de búsqueda que permite al usuario buscar un contacto y, si lo encuentra, muestra la dirección de ese contacto en un mapa. La primera versión de esta aplicación demuestra cómo agregar contactos a la base de datos, y cómo mostrar la lista de contactos en un componente JSF **Tabla**. En la segunda versión, agregamos un componente **Auto Complete Text Field** habilitado para Ajax y lo habilitamos para sugerir una lista de nombres de contactos, a medida que el usuario escribe información. La última versión nos permite buscar un contacto en la libreta de direcciones, y mostrar la dirección correspondiente en un mapa mediante el uso del componente `MapViewer` habilitado para Ajax, controlado mediante Google Maps (`maps.google.com`).

Al igual que en el capítulo 26, desarrollamos los ejemplos de este capítulo en Java Studio Creator 2.0. Instalamos una biblioteca de componentes suplementaria (la **biblioteca de componentes Ajax de Java BluePrints**), la cual proporciona los componentes habilitados para Ajax que utilizamos en la aplicación de libreta de direcciones. En la sección 27.3.1 se incluyen instrucciones para instalar esta biblioteca.

27.2 Acceso a bases de datos en las aplicaciones Web

Muchas aplicaciones Web acceden a bases de datos para almacenar y obtener datos persistentes. En esta sección vamos a crear una aplicación Web que utiliza una base de datos Java DB para almacenar contactos en la libreta de direcciones y mostrar contactos de esta libreta en una página Web.

La página Web permite al usuario introducir nuevos contactos en un formulario. Este formulario consiste en componentes **Campo de texto** para el primer nombre del contacto, su dirección física, ciudad, estado y códi-

go postal. El formulario también tiene un botón **Enviar** para enviar los datos al servidor, y un botón **Clear** para restablecer los campos del formulario. La aplicación almacena la información de la libreta de direcciones en una base de datos llamada `LibretaDirecciones`, la cual tiene una sola tabla llamada `Addresses`. (En el directorio de ejemplos para este capítulo proporcionamos esta base de datos. Puede descargar los ejemplos de `www.deitel.com/books/jhtp7`). Este ejemplo también introduce el componente JSF **Tabla**, el cual muestra las direcciones de la base de datos en formato tabular. En breve le mostraremos cómo configurar el componente **Tabla**.

27.2.1 Creación de una aplicación Web que muestra datos de una base de datos

Ahora le explicaremos cómo crear la GUI de la aplicación `LibretaDirecciones` y establecer un enlace de datos que permita al componente **Tabla** mostrar información de la base de datos. Más adelante en esta sección presentaremos el archivo JSP generado, y hablaremos sobre el archivo de bean de página relacionado en la sección 27.2.2. Para crear la aplicación `LibretaDirecciones`, realice los siguientes pasos:

Paso 1: Crear el proyecto

En Java Studio Creator 2, cree un proyecto **Aplicación web JSF** llamado `LibretaDirecciones`. Cambie el nombre a los archivos JSP y de bean de página a `LibretaDirecciones`, usando las herramientas de refactorización.

Paso 2: Crear el formulario para la entrada del usuario

En modo **Diseño**, agregue un componente **Texto estático** a la parte superior de la página que contenga el texto `"Agregar un contacto a la libreta de direcciones:"` y utilice la propiedad `style` del componente para establecer el tamaño de la fuente en `18px`. Agregue seis componentes **Campo de texto** a la página y cambie su nombre a `pnombreCampoTexto`, `apaternoCampoTexto`, `calleCampoTexto`, `ciudadCampoTexto`, `estadoCampoTexto` y `cpCampoTexto`. Establezca la propiedad `required` de cada **Campo de texto** en `true`; para ello seleccione el componente **Campo de texto** y después haga clic sobre la casilla de verificación de la propiedad `required`. Etiquete cada **Campo de texto** con un componente **Etiqueta** y asocie ese componente con su correspondiente **Campo de texto**. Por último, agregue los botones **Enviar** y **Borrar**. Establezca la propiedad `primary` del botón **Enviar** a `true`, para hacer que resalte más en la página que el botón **Borrar**, y para permitir que el usuario envíe un nuevo contacto, al oprimir *Intro* en vez de hacer clic en el botón **Enviar**. Establezca la propiedad `reset` del botón a `true`, para evitar la validación cuando el usuario haga clic en el botón **Borrar**. Como vamos a borrar los campos, no deseamos asegurarnos que contengan información. Hablaremos sobre el manejador de acciones para el botón **Enviar** después de presentar el archivo de bean de página. El botón **Borrar** no necesita un método manejador de acciones, ya que al establecer la propiedad `reset` a `true` el botón se configura de manera automática para restablecer todos los campos de entrada de la página. Cuando haya terminado estos pasos, su formulario deberá verse como el de la figura 27.1.

Paso 3: Agregar un componente Tabla a la página

Arrastre un componente **Tabla** de la sección **Básicos** de la **Paleta** a la página, y colóquelo justo debajo de los dos componentes **Botón**. Cambie su nombre a `direccionesTabla`. El componente **Tabla** da formato y muestra

Figura 27.1 | Formulario de la aplicación `LibretaDirecciones` para agregar un contacto.

datos de las tablas de una base de datos. En la ventana **Propiedades**, cambie la propiedad `title` de **Tabla** a `Contactos`. En breve le mostraremos cómo configurar la **Tabla** para que interactúe con la base de datos `LibretaDirecciones`.

Paso 4: Cómo agregar una base de datos a una aplicación Web de Java Studio Creator 2

Para este ejemplo, utilizaremos una base de datos Java DB llamada `LibretaDirecciones` con una sola tabla llamada `Addresses`. Para que esta base de datos esté disponible en sus proyectos, copie la carpeta `LibretaDirecciones` de la carpeta de ejemplos del capítulo, a la carpeta `SunAppServer8\derby\databases` de la carpeta de instalación de Java Studio Creator 2.

Para utilizar una base de datos en una aplicación Web de Java Studio Creator 2, primero debemos iniciar el **servidor de bases de datos integrado** del IDE, el cual permite utilizar conexiones a bases de datos en los proyectos de Java Studio Creator 2. El servidor incluye controladores para muchas bases de datos, incluyendo Java DB. Haga clic en la ficha **Servidores** debajo del menú **Archivo**, haga clic con el botón derecho en **Servidor Bundled Database** en la parte inferior de la ventana **Servidores** y seleccione **Iniciar Bundled Database**. Ahora podrá utilizar bases de datos que se ejecuten en este servidor en sus aplicaciones.

Para agregar la base de datos `LibretaDirecciones` a este proyecto, haga clic con el botón derecho en el nodo **Orígenes de datos** en la parte superior de la ventana **Servidores** y seleccione **Agregar origen de datos...**. En el cuadro de diálogo **Agregar origen de datos** (figura 27.2), escriba `LibretaDirecciones` para el nombre del origen de datos y seleccione **Derby** en el tipo de servidor. (En el capítulo 25 vimos que Java DB es la versión producida por Sun de Apache Derby). El ID de usuario y la contraseña para esta base de datos son `jhtp7`. Para el URL de la base de datos, escriba `jdbc:derby://localhost:21527/LibretaDirecciones`. Este URL indica que la base de datos reside en el equipo local y acepta conexiones en el puerto 21527. Haga clic en el botón **Seleccionar** para elegir una tabla que se utilizará para validar la base de datos. En el cuadro de diálogo que aparezca, seleccione la tabla **JHTP7.ADDRESSES**, ya que es la única tabla en la base de datos. Haga clic en **Seleccionar** para cerrar este cuadro de diálogo, y después haga clic en **Agregar** para agregar la base de datos como origen de datos para el proyecto y cierre el cuadro de diálogo. [*Nota:* Java Studio Creator 2 muestra los nombres de las bases de datos y las tablas en mayúsculas].

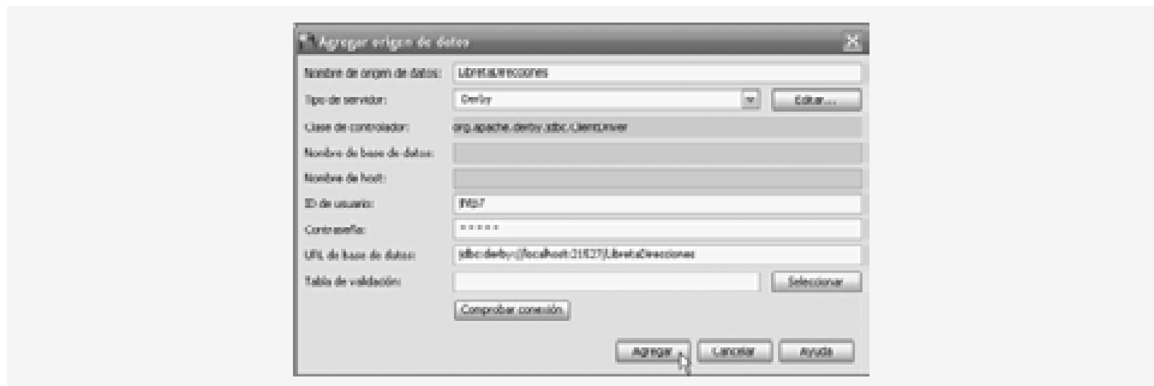


Figura 27.2 | Cuadro de diálogo para agregar un origen de datos.

Paso 5: Enlazar el componente Tabla a la tabla `Addresses` *de la base de datos* `LibretaDirecciones`

Ahora que hemos configurado un origen de datos para la tabla `Addresses` de la base de datos, podemos configurar el componente **Tabla** para mostrar los datos de `LibretaDirecciones`. Simplemente arrastre la tabla de la base de datos de la ficha Servidores, y suéltela en el componente **Tabla** para crear el enlace.

Si necesita un control más preciso sobre las columnas a mostrar, puede enlazar con una tabla de la base de datos de la siguiente manera: haga clic con el botón derecho del ratón en el componente **Tabla** y seleccione **Enlazar con datos** para mostrar el cuadro de diálogo **Diseño de tabla**. Haga clic en el botón **Agregar proveedor de datos...** para mostrar el cuadro de diálogo **Agregar proveedor de datos**, el cual contiene una lista de los orígenes de datos disponibles. Expanda el nodo `LibretaDirecciones`, expanda el nodo **Tablas**, seleccione **ADDRESSES** y haga clic

en **Agregar**. Ahora el cuadro de diálogo **Diseño de tabla** mostrará una lista de las columnas en la tabla `Addresses` de la base de datos (figura 27.3). Todos los elementos bajo el encabezado **Seleccionado** se mostrarán en la **Tabla**. Para eliminar una columna de la **Tabla**, puede seleccionarla y hacer clic en el botón **<**. Como deseamos mostrar todas estas columnas en nuestra tabla, simplemente haga clic en **Aceptar** para salir del cuadro de diálogo.

De manera predeterminada, la **Tabla** utiliza los nombres de las columnas de la tabla de la base de datos en mayúsculas como encabezados. Para modificar estos encabezados, seleccione una columna y edite su propiedad `headerText` en la ventana **Propiedades**. Para seleccionar una columna, expanda el nodo `direccionesTabla` en la ventana **Esquema** (estando en modo **Diseño**) y después seleccione el objeto columna apropiado. También modificamos la propiedad `id` de cada columna, para hacer más legibles los nombres de las variables en el código. En modo **Diseño**, los encabezados de las columnas de su **Tabla** deberán aparecer como en la figura 27.4.

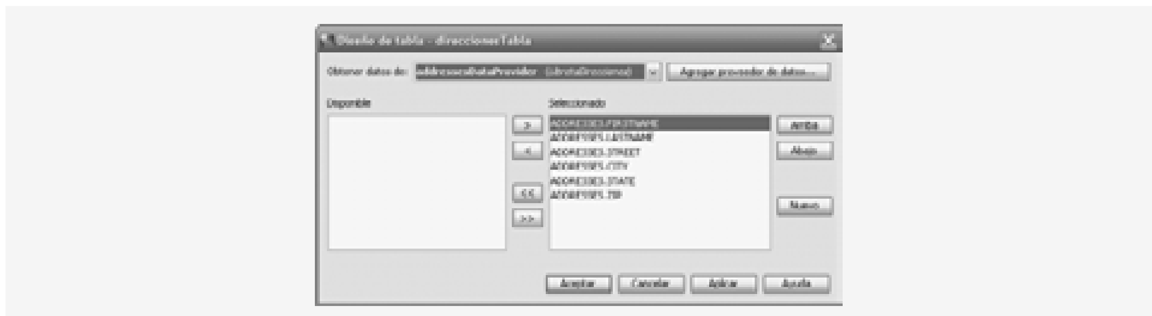


Figura 27.3 | Cuadro de diálogo para enlazar con la tabla `Addresses`.

Contactos					
Primer nombre	Apellido paterno	Calle	Ciudad	Estado	CP
abc	abc	abc	abc	abc	abc
abc	abc	abc	abc	abc	abc
abc	abc	abc	abc	abc	abc

Figura 27.4 | El componente `Tabla` después de enlazarlo con una tabla de la base de datos y editar los nombres de su columna, para fines de visualización.

Una libreta de direcciones podría contener muchos contactos, por lo que sería conveniente mostrar sólo unos cuantos a la vez. Al hacer clic en la casilla de verificación a un lado de la propiedad `paginationControl` de la tabla en la ventana **Propiedades**, se configura esta **Tabla** para paginación automática. Se mostrarán cinco filas a la vez, y se agregarán botones para avanzar hacia delante y hacia atrás, entre grupos de cinco contactos, al final de la **Tabla**. (También puede usar la ficha **Opciones** del cuadro de diálogo **Diseño de tabla** para seleccionar la paginación y el número de filas. Para ver esta ficha, haga clic con el botón derecho en la **Tabla**, seleccione **Diseño de página...**, y después haga clic en la ficha **Opciones**). A continuación, establezca la propiedad `internalVirtualForm` de `direccionesTabla`. Los formularios virtuales permiten enviar subconjuntos de los componentes de entrada de un formulario al servidor. Al establecer esta propiedad se evita que los botones de control de paginación en la **Tabla** envíen los componentes **Campo de texto** en el formulario cada vez que el usuario desea ver el siguiente grupo de contactos. En la sección 27.4.1 hablaremos sobre los formularios virtuales.

Observe que al enlazar la **Tabla** con un proveedor de datos, se agregó un nuevo objeto `addressesDataprovider` (una instancia de la clase **`CachedRowSetDataProvider`**) al nodo **LibretaDirecciones** en la ventana **Esquema**. Un objeto `CachedRowSetDataProvider` proporciona un objeto `RowSet` desplazable que puede enlazarse con un componente **Tabla** para mostrar los datos del objeto `RowSet`. Este proveedor de datos es una envoltura para

un objeto `CachedRowSet`. Si hace clic en el elemento **addressesDataProvider** en la ventana **Esquema**, podrá ver en la ventana **Propiedades** que su propiedad `cachedRowSet` se estableció en `addressesRowSet`, un objeto que implementa a la interfaz `CachedRowSet`.

Paso 6: Modificar la instrucción SQL de `addressesRowSet`

El objeto `CachedRowSet` envuelto por nuestro objeto `addressesDataProvider` está configurado de manera predeterminada para ejecutar una consulta SQL que seleccione todos los datos en la tabla `Direcciones` de la base de datos `LibretaDirecciones`. Para editar esta consulta SQL, puede expandir el nodo `SessionBean` en la ventana **Esquema** y hacer doble clic en el elemento `addressesRowSet` para abrir la ventana del editor de consultas (figura 27.5). Nos gustaría editar la instrucción SQL de manera que los registros con apellidos duplicados se ordenen por apellido, y después por primer nombre. Para ello, haga clic en la columna **Tipo de orden** enseguida de la fila **LASTNAME** y seleccione **Ascendente**. Después, repita esto para la fila **FIRSTNAME**. Observe que la expresión

```
ORDER BY JHTP7.ADDRESSES.LASTNAME ASC,
    JHTP7.ADDRESSES.FIRSTNAME ASC
```

se agregó a la instrucción SQL al final del editor.

Paso 7: Agregar validación

Es importante validar los datos del formulario en esta página, para asegurar que los datos puedan insertarse correctamente en la base de datos `LibretaDirecciones`. Todas las columnas de la base de datos son de tipo `varchar` y tienen restricciones de longitud. Por esta razón, debemos agregar un **Validador de longitud** a cada componente **Campo de texto**, o establecer la propiedad `maxLength` de cada componente **Campo de texto**. Optamos por establecer la propiedad `maxLength` de cada uno. Los componentes **Campo de texto** del primer nombre, apellido paterno, calle, ciudad, estado y código postal no pueden exceder a 20, 30, 100, 30, 2 y 5 caracteres, respectivamente.

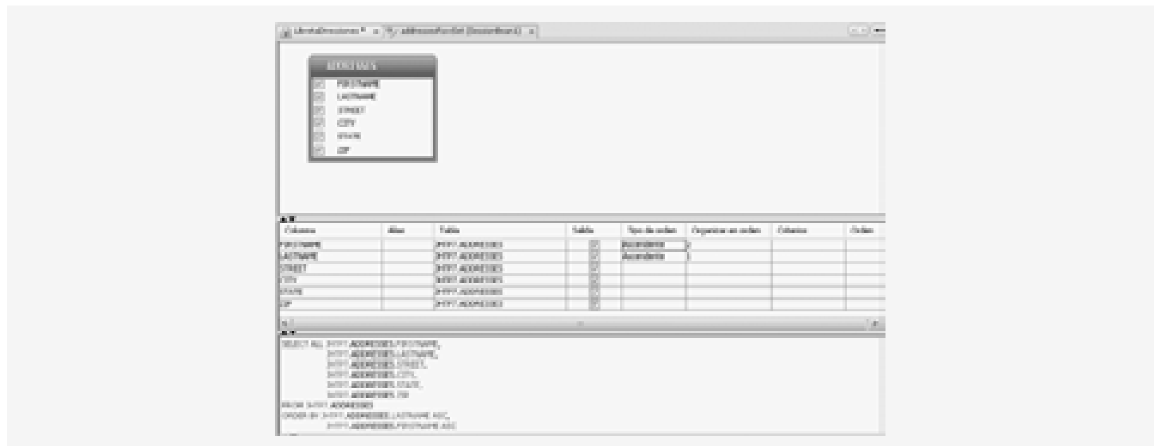


Figura 27.5 | Edición de la instrucción SQL de `addressesRowSet`.

Por último, arrastre un componente **Grupo de mensaje** a su página, a la derecha de la **Tabla**. Un componente **Grupo de mensaje** muestra mensajes del sistema. Utilizamos este componente para mostrar un mensaje de error cuando falla un intento de contactarse con la base de datos. Establezca la propiedad `showGlobalOnly` del **Grupo de mensaje** a `true`, para evitar que se muestren aquí mensajes de error de validación a nivel de componente.

Archivo JSP para una página Web que interactúa con una base de datos

El archivo JSP para la aplicación se muestra en la figura 27.6. Este archivo contiene una gran cantidad de marcado generado para los componentes que vimos en el capítulo 26. En este ejemplo sólo hablaremos sobre el marcado para los componentes que son nuevos.

```
<?xml version="1.0" encoding="UTF-8"?>

<!-- Fig. 27.6: LibretaDirecciones.jsp -->
<!-- JSP de LibretaDirecciones con un formulario para agregar y un componente JSF Tabla
-->

<jsp:root version="1.2" xmlns:f="http://java.sun.com/jsf/core"
   xmlns:h="http://java.sun.com/jsf/html" xmlns:jsp=
   "http://java.sun.com/JSP/Page" xmlns:ui="http://www.sun.com/web/ui">
   <jsp:directive.page contentType="text/html;charset=UTF-8"
      pageEncoding="UTF-8"/>
   <f:view>
      <ui:page binding="#{LibretaDirecciones.page1}" id="page1">
         <ui:html binding="#{LibretaDirecciones.html1}" id="html1">
            <ui:head binding="#{LibretaDirecciones.head1}" id="head1">
               <ui:link binding="#{LibretaDirecciones.link1}" id="link1"
                  url="/resources/stylesheet.css"/>
         </ui:head>
         <ui:body binding="#{LibretaDirecciones.body1}" id="body1"
            style="-rave-layout: grid">
            <ui:form binding="#{LibretaDirecciones.form1}" id="form1">
               <ui:staticText binding="#{LibretaDirecciones.staticText1}" id=
                  "staticText1" style="font-size: 18px; left: 12px;
                  top: 24px; position: absolute"
                  text="Agregar un contacto a la libreta de direcciones:"/>
               <ui:textField binding="#{LibretaDirecciones.pnombreCampoTexto}"
                  id="pnombreCampoTexto" maxLength="20" required="true"
                  style="left: 132px; top: 72px;
                  position: absolute"/>
               <ui:textField binding="#{LibretaDirecciones.apaternoCampoTexto}"
                  id="apaternoCampoTexto" maxLength="30" required="true"
                  style="left: 504px; top: 72px; position: absolute;
                  width: 228px"/>
               <ui:textField binding="#{LibretaDirecciones.calleCampoTexto}"
                  id="calleCampoTexto" maxLength="100" required="true"
                  style="left: 132px; top: 96px; position: absolute;
                  width: 600px"/>
               <ui:textField binding="#{LibretaDirecciones.ciudadCampoTexto}"
                  id="ciudadCampoTexto" maxLength="30" required="true"
                  style="left: 132px; top: 120px; position: absolute; width: 264px"/>
               <ui:textField binding="#{LibretaDirecciones.estadoCampoTexto}"
                  id="estadoCampoTexto" maxLength="2" required="true"
                  style="left: 480px; top: 120px; position: absolute;
                  width: 60px"/>
               <ui:textField binding="#{LibretaDirecciones.cpCampoTexto}"
                  id="cpCampoTexto" maxLength="5" required="true"
                  style="left: 672px; top: 120px; position: absolute;
                  width: 60px"/>
               <ui:label binding="#{LibretaDirecciones.pnombreEtiqueta}" for=
                  "pnombreCampoTexto" id="pnombreEtiqueta" style=position: absolute"
                  "left: 12px; top: 72px; text="Primer nombre:"/>
               <ui:label binding="#{LibretaDirecciones.apaternoEtiqueta}" for=
                  "apaternoCampoTexto" id="apaternoEtiqueta" style="position: absolute;
                  left: 384px; top: 72px" text="Apellido Paterno:"/>
               <ui:label binding="#{LibretaDirecciones.calleEtiqueta}" for=
                  "calleCampoTexto" id="calleEtiqueta" style=position: absolute"
                  "left: 12px; top: 96px;  text="Calle:"/>
               <ui:label binding="#{LibretaDirecciones.ciudadEtiqueta}" for=
                  "ciudadCampoTexto" id="ciudadEtiqueta" style="left: 12px;
```

Figura 27.6 | JSP de `LibretaDirecciones` con un formulario para agregar y un componente JSF **Tabla**. (Parte 1 de 4).

```
                  top: 120px; position: absolute" text="Ciudad:"/>
               <ui:label binding="#{LibretaDirecciones.estadoEtiqueta}" for=
                  "estadoCampoTexto" id="estadoEtiqueta" style= position: absolute"
                  "left: 408px; top: 120px;  text="Estado:"/>
               <ui:label binding="#{LibretaDirecciones.cpEtiqueta}" for=
                  "cpCampoTexto" id="cpEtiqueta" style="height: 22px; left: 552px;
                  top: 120px; position: absolute; width: 94px" text="Código postal:"/>
               <ui:button action="#{LibretaDirecciones.enviarBoton_action}"
                  binding="#{LibretaDirecciones.enviarBoton}" id=
                  "enviarBoton" primary="true" style= position: absolute"
                  "left: 131px; top: 168px; text="Enviar"/>
               <ui:button binding="#{LibretaDirecciones.borrarBoton}" id=
                  "borrarBoton" reset="true" style="left: 251px; top:
                  168px; position: absolute" text="Borrar"/>
               <ui:table augmentTitle="false" binding=
                  "#{LibretaDirecciones.direccionesTabla}" id="direccionesTabla"
                  paginationControls="rue" style="left: 12px; top: 204px;
                  position: absolute; width: 720px"
                  title="Contactos" width="720">
               <script><![CDATA[
<!--Las líneas 79 a 140 contienen código de JavaScript que se eliminó para ahorrar espacio.
   El código fuente completo se proporciona en la carpeta de este ejemplo. -->
               }]]></script>
                  <ui:tableRowGroup binding=
                     "#{LibretaDirecciones.tableRowGroup1}" id=
                     "tableRowGroup1" rows="5" sourceData=
                     "#{LibretaDirecciones.addressesDataProvider}"
                     sourceVar="currentRow">
                     <ui:tableColumn binding=
                        "#{LibretaDirecciones.pnombreColumna}" headerText=
                        "Primer nombre" id="pnombreColumna" sort=
                        "ADDRESSES.FIRSTNAME">
                        <ui:staticText binding=
                           "#{LibretaDirecciones.staticText2}" id=
                           "staticText2" text="#{currentRow.value[
                           'ADDRESSES.FIRSTNAME']}"/>
                     </ui:tableColumn>
                     <ui:tableColumn binding=
                        "#{LibretaDirecciones.apaternoColumna}" headerText=
                        "Apellido paterno" id="apaternoColumna"
                        sort="ADDRESSES.LASTNAME">
                        <ui:staticText binding=
                           "#{LibretaDirecciones.staticText3}" id=
                           "staticText3" text="#{currentRow.value[
                           'ADDRESSES.LASTNAME']}"/>
                     </ui:tableColumn>
                     <ui:tableColumn binding=
                        "#{LibretaDirecciones.calleColumna}" headerText=
                        "Calle" id="calleColumna"
                        sort="ADDRESSES.STREET">
                        <ui:staticText binding=
                           "#{LibretaDirecciones.staticText4}" id=
                           "staticText4" text="#{currentRow.value[
                           'ADDRESSES.STREET']}"/>
                     </ui:tableColumn>
                     <ui:tableColumn binding=
                        "#{LibretaDirecciones.ciudadColumna}" headerText="Ciudad"
                        id="ciudadColumna" sort="ADDRESSES.CITY">
                        <ui:staticText binding=
```

Figura 27.6 | JSP de `LibretaDirecciones` con un formulario para agregar y un componente JSF **Tabla**. (Parte 2 de 4).

```
                              "#{LibretaDirecciones.staticText5}" id="staticText5"
                              text="#{currentRow.value[
                              'ADDRESSES.CITY']}"/>
                        </ui:tableColumn>
                        <ui:tableColumn binding=
                           "#{LibretaDirecciones.estadoColumna}" headerText="Estado"
                           id="estadoColumna" sort="ADDRESSES.STATE">
                           <ui:staticText binding=
                              "#{LibretaDirecciones.staticText6}" id=
                              "staticText6" text="#{currentRow.value[
                              'ADDRESSES.STATE']}"/>
                           </ui:tableColumn>
                           <ui:tableColumn binding=
                              "#{LibretaDirecciones.cpColumna}" headerText="CP"
                              id="cpColumna" sort="ADDRESSES.ZIP">
                              <ui:staticText binding=
                                 "#{LibretaDirecciones.staticText7}" id="staticText7"
                                 text="#{currentRow.value[
                                 'ADDRESSES.ZIP']}"/>
                           </ui:tableColumn>
                        </ui:tableRowGroup>
                     </ui:table>
                     <ui:messageGroup binding="#{LibretaDirecciones.messageGroup1}"
                        id="messageGroup1" showGlobalOnly="true" style=
                        "left: 744px; top: 204px; position: absolute"/>
               </ui:form>
            </ui:body>
         </ui:html>
      </ui:page>
   </f:view>
</jsp:root>
```

a)



Figura 27.6 | JSP de `LibretaDirecciones` con un formulario para agregar y un componente JSF **Tabla**. (Parte 3 de 4).

b)

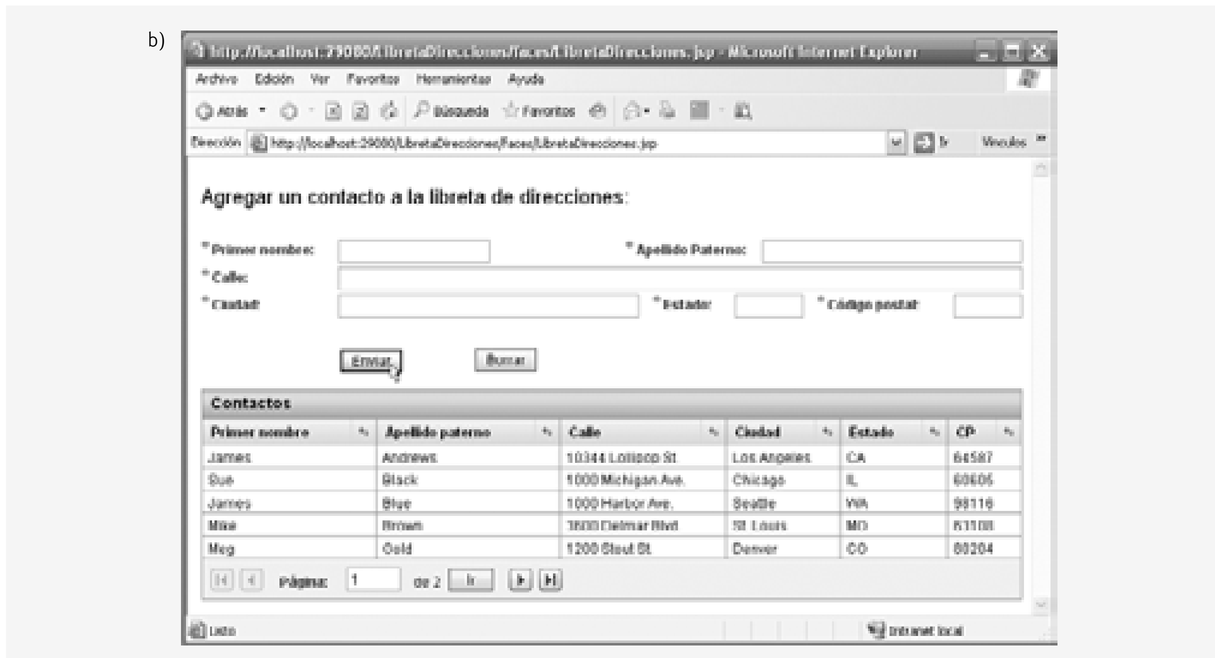


Figura 27.6 | JSP de `LibretaDirecciones` con un formulario para agregar y un componente JSF **Tabla**. (Parte 4 de 4).

Las líneas 21 a 72 contienen los componentes JSF que conforman el formulario que recopila la entrada del usuario. En las líneas 73 a 199 se define el elemento **Tabla** (`ui:table`) que muestra la información de las direcciones de la base de datos. Las líneas 79 a 140 (que no se muestran aquí) contienen funciones de JavaScript generadas por el IDE para manejar las acciones de **Tabla**, como un cambio en el estado de la fila actual. Los componentes JSF **Tabla** pueden tener varios grupos de filas que muestren distintos datos. Esta **Tabla** tiene un solo elemento `ui:tableRowGroup` con una marca inicial en las líneas 142 a 146. El atributo `sourceData` del grupo de filas está enlazado a nuestro objeto `addressesDataProvider` y recibe el nombre de variable `currentRow`. El grupo de filas también define las columnas de la **Tabla**. Cada elemento `ui:tableColumn` contiene un elemento `ui:staticText` con su atributo `text` enlazado con una columna en la fila actual (`currentRow`) del proveedor de datos. Estos elementos `ui:staticText` permiten a la `Tabla` mostrar los datos de cada fila.

Bean de sesión para la aplicación `LibretaDirecciones`

En la figura 27.7 muestra el archivo `SessionBean1.java` generado por Java Studio Creator 2 para la aplicación `LibretaDirecciones`. El objeto `CachedRowSet` que utiliza el proveedor de datos del componente **Tabla** para acceder a la base de datos `LibretaDirecciones` es una propiedad de esta clase (líneas 31 a 41).

```
1  // Fig. 27.7: SessionBean1.java
2  // Bean de sesión que inicializa el origen de datos para la
3  // base de datos LibretaDirecciones.
4  package libretadirecciones;
5
6  import com.sun.rave.web.ui.appbase.AbstractSessionBean;
7  import javax.faces.FacesException;
8  import com.sun.sql.rowset.CachedRowSetXImpl;
9
10 public class SessionBean1 extends AbstractSessionBean
11 {
```

Figura 27.7 | Bean de sesión que inicializa el origen de datos para la base de datos `LibretaDirecciones`. (Parte 1 de 2).

```
12      private int __placeholder;
13
14      private void _init() throws Exception
15      {
16         addressesRowSet.setDataSourceName(
17            "java:comp/env/jdbc/LibretaDirecciones" );
18         addressesRowSet.setCommand(
19            "SELECT ALL JHTP7.ADDRESSES.FIRSTNAME," +
20            "\nJHTP7.ADDRESSES.LASTNAME," +
21            "\nJHTP7.ADDRESSES.STREET," +
22            "\nJHTP7.ADDRESSES.CITY," +
23            "\nJHTP7.ADDRESSES.STATE," +
24            "\nJHTP7.ADDRESSES.ZIP" +
25            "\nFROM JHTP7.ADDRESSES" +
26            "\nORDER BY JHTP7.ADDRESSES.LASTNAME ASC," +
27            "\nJHTP7.ADDRESSES.FIRSTNAME ASC " );
28         addressesRowSet.setTableName( "ADDRESSES" );
29      } // fin del método _init
30
31      private CachedRowSetXImpl addressesRowSet = new CachedRowSetXImpl();
32
33      public CachedRowSetXImpl getAddressesRowSet()
34      {
35         return addressesRowSet;
36      }
37
38      public void setAddressesRowSet(CachedRowSetXImpl crsxi)
39      {
40         this.addressesRowSet = crsxi;
41      }
42
43      // Las líneas 43 a 76 del código generado en forma automática se eliminaron para
           ahorrar espacio.
44      // El código fuente completo se proporciona en la carpeta de este ejemplo.
45   } // fin de la clase SessionBean1
```

Figura 27.7 | Bean de sesión que inicializa el origen de datos para la base de datos `LibretaDirecciones`. (Parte 2 de 2).

El método `_init` (líneas 14 a 29) configura a `addressesRowSet` para que interactúe con la base de datos `LibretaDirecciones` (líneas 16 a 27). En las líneas 16 y 17 se conecta el conjunto de filas con la base de datos. En las líneas 18 a 27 se establece el comando SQL de `addressesRowSet` con la consulta configurada en la figura 27.5.

27.2.2 Modificación del archivo de bean de página para la aplicación `LibretaDirecciones`

Después de crear la página Web y configurar los componentes utilizados en este ejemplo, haga doble clic en el botón **Enviar** para crear un manejador de eventos de acción para este botón en el archivo de bean de página. El código para insertar un contacto en la base de datos se colocará en este método. El bean de página con el manejador de eventos completo se muestra en la figura 27.8 a continuación.

Las líneas 534 a 573 contienen el código para manejar los eventos del botón **Enviar**. En la línea 536 se determina si se puede anexar una nueva fila al proveedor de datos. De ser así, se anexa una nueva fila en la línea 540. Cada fila en un objeto `CachedRowSetDataProvider` tiene su propia clave; el método **appendRow** devuelve la clave para la nueva fila. En la línea 541 se establece el cursor del proveedor de datos a la nueva fila, de manera que cualquier modificación que realicemos al proveedor de datos afecte a esa fila. En las líneas 543 a 554 se establece cada una de las columnas de la fila a los valores introducidos por el usuario en los correspondientes componentes **Campo de texto**. En la línea 555 se almacena el nuevo contacto, llamando al método **`commitChanges`** de la clase `CachedRowSetDataProvider` para insertar la nueva fila en la base de datos `LibretaDirecciones`.

```
// Fig. 27.8: LibretaDirecciones.java
// Bean de página para agregar un contacto a la libreta de direcciones.
package libretadirecciones;

import com.sun.data.provider.RowKey;
import com.sun.rave.web.ui.appbase.AbstractPageBean;
import com.sun.rave.web.ui.component.Body;
import com.sun.rave.web.ui.component.Form;
import com.sun.rave.web.ui.component.Head;
import com.sun.rave.web.ui.component.Html;
import com.sun.rave.web.ui.component.Link;
import com.sun.rave.web.ui.component.Page;
import javax.faces.FacesException;
import com.sun.rave.web.ui.component.StaticText;
import com.sun.rave.web.ui.component.TextField;
import com.sun.rave.web.ui.component.Label;
import com.sun.rave.web.ui.component.Button;
import com.sun.rave.web.ui.component.Table;
import com.sun.rave.web.ui.component.TableRowGroup;
import com.sun.rave.web.ui.component.TableColumn;
import com.sun.data.provider.impl.CachedRowSetDataProvider;
import com.sun.rave.web.ui.component.MessageGroup;

public class LibretaDirecciones extends AbstractPageBean
{
   private int __placeholder;

   private void _init() throws Exception
   {
      addressesDataProvider.setCachedRowSet(
         (javax.sql.rowset.CachedRowSet)
            getValue("#{SessionBean1.addressesRowSet}"));
      direccionesTabla.setInternalVirtualForm(true);
   }

   // Las líneas 36 a 521 del código generado en forma automática se eliminaron para
   ahorrar espacio.
   // El código fuente completo se proporciona en la carpeta de este ejemplo.

   public void prerender()
   {
      addressesDataProvider.refresh();
   } // fin del método prerender

   public void destroy()
   {
      addressesDataProvider.close();
   } // fin del método destroy

   // manejador de acciones que agrega un contacto a la base de datos LibretaDirecciones
   // cuando el usuario hace clic en el botón Enviar
   public String enviarBoton_action()
   {
      if ( addressesDataProvider.canAppendRow() )
   {
         try
         {
            RowKey rk = addressesDataProvider.appendRow();
            addressesDataProvider.setCursorRow(rk);
```

Figura 27.8 | Bean de página para agregar un contacto a la libreta de direcciones. (Parte 1 de 2).

```
542
543             addressesDataProvider.setValue( "ADDRESSES.FIRSTNAME",
544                pnombreCampoTexto.getValue() );
545             addressesDataProvider.setValue( "ADDRESSES.LASTNAME",
546                apaternoCampoTexto.getValue() );
547             addressesDataProvider.setValue( "ADDRESSES.STREET",
548                calleCampoTexto.getValue() );
549             addressesDataProvider.setValue( "ADDRESSES.CITY",
550                ciudadCampoTexto.getValue() );
551             addressesDataProvider.setValue( "ADDRESSES.STATE",
552                estadoCampoTexto.getValue() );
553             addressesDataProvider.setValue( "ADDRESSES.ZIP",
554                cpCampoTexto.getValue());
555             addressesDataProvider.commitChanges();
556
557             // restablece los campos de texto
558             apaternoCampoTexto.setValue( "" );
559             pnombreCampoTexto.setValue( "" );
560             calleCampoTexto.setValue( "" );
561             ciudadCampoTexto.setValue( "" );
562             estadoCampoTexto.setValue( "" );
563             cpCampoTexto.setValue( "" );
564          } // fin de try
565          catch ( Exception ex )
566          {
567             error( "No se actualizo la libreta de direcciones.  " +
568                ex.getMessage() );
569          } // fin de catch
570       } // fin de if
571
572          return null;
573     } // fin del método enviarBoton_action
574  } // fin de la clase LibretaDirecciones
```

Figura 27.8 | Bean de página para agregar un contacto a la libreta de direcciones. (Parte 2 de 2).

En las líneas 558 a 563 se borran todos los componentes **Campo de texto** del formulario. Si se omiten estas líneas, los campos retendrán sus valores actuales una vez que se actualice la base de datos y se vuelva a cargar la página. Además, el botón **Borrar** no trabajará en forma apropiada si no se borran los componentes **Campo de texto**. En vez de vaciar los componentes **Campo de texto**, los restablecerá a los valores que contenían la última vez que se envió el formulario.

En las líneas 565 a 569 se atrapan las excepciones que podrían ocurrir mientras se realiza la actualización de la base de datos `LibretaDirecciones`. En las líneas 567 y 568 se muestra un mensaje indicando que la base de datos no se actualizó, así como el mensaje de error de la excepción, en el componente `MessageGroup` de la página.

En el método `prerender`, en la línea 524 se hace una llamada al método **`refresh`** de `CachedRowSetDataProvider`. Esto vuelve a ejecutar la instrucción SQL del objeto `CachedRowSet` envuelto y se vuelven a ordenar las filas de la **Tabla**, de manera que la nueva fila se muestre en el orden apropiado. Si no se hace la llamada a `refresh`, la nueva dirección se mostrará al final de la **Tabla** (ya que anexamos la nueva fila al final del proveedor de datos). El IDE generó código automáticamente para liberar los recursos utilizados por el proveedor de datos (línea 529) en el método `destroy`.

27.3 Componentes JSF habilitados para Ajax

El término **Ajax (JavaScript y XML asíncronos)** fue ideado por Jesse James Garrett de Adaptive Path, Inc. en febrero de 2005, para describir un rango de tecnologías para desarrollar aplicaciones Web dinámicas y con gran capacidad de respuesta. Las aplicaciones Ajax incluyen Google Maps, FlickR de Yahoo y muchas más. Ajax separa la parte correspondiente a la interacción con el usuario de una aplicación, de la interacción con su servidor, permitiendo que ambas procedan en forma asincrónica y en paralelo. Esto permite a las aplicaciones Ajax basadas en

Web ejecutarse a velocidades que se asemejan a las de las aplicaciones de escritorio, con lo cual se reduce (o incluso se elimina) la tradicional ventaja de rendimiento que han tenido las aplicaciones de escritorio en comparación con las aplicaciones Web. Esto implica enormes ramificaciones para la industria de las aplicaciones de escritorio; la plataforma preferida para las aplicaciones está empezando a cambiar, del escritorio a la Web. Muchas personas creen que la Web (en especial, dentro del contexto del abundante software de código fuente abierto, las computadoras económicas y el explosivo incremento en el ancho de banda de Internet) creará la siguiente fase principal de crecimiento para las compañías de Internet.

Ajax realiza llamadas asíncronas al servidor para intercambiar pequeñas cantidades de datos con cada llamada. En donde, por lo general, la página completa se enviaría y se volvería a cargar con cada interacción del usuario en una página Web, Ajax permite que se vuelvan a cargar sólo las porciones necesarias de la página, lo cual ahorra tiempo y recursos.

Las aplicaciones Ajax contienen marcado de XHTML y CSS al igual que cualquier otra página Web, y hacen uso de las tecnologías de secuencias de comandos del lado cliente (como JavaScript) para interactuar con los elementos de las páginas. El objeto `XMLHttpRequestObject` permite los intercambios asíncronos con el servidor Web, lo cual hace que las aplicaciones Ajax tengan una gran capacidad de respuesta. Este objeto se puede utilizar en la mayoría de los lenguajes de secuencias de comandos para pasar datos XML del cliente al servidor, y para procesar los datos XML que el servidor envía de vuelta al cliente.

Aunque el uso de las tecnologías Ajax en las aplicaciones Web puede mejorar en forma considerable el rendimiento, la programación en Ajax es compleja y propensa a errores. Los diseñadores de páginas requieren conocer tanto los lenguajes de secuencias de comandos como los lenguajes de marcado. Las **bibliotecas Ajax** facilitan el proceso de aprovechar los beneficios de Ajax en las aplicaciones Web, sin tener que escribir Ajax "puro". Estas bibliotecas proporcionan elementos de página habilitados para Ajax, que pueden incluirse en las páginas Web con sólo agregar al marcado de la página las etiquetas definidas en la biblioteca. Limitaremos nuestra discusión acerca de cómo crear aplicaciones Ajax al uso de una de esas bibliotecas en Java Studio Creator 2.

27.3.1 Biblioteca de componentes Java BluePrints

La biblioteca de componentes **Java BluePrints** de Ajax proporciona **componentes JSF habilitados para Ajax**. Estos componentes dependen de la tecnología Ajax para brindar la sensación y capacidad de respuesta de una aplicación de escritorio a través de la Web. En la figura 27.9 se muestra un resumen del conjunto actual de componentes que podemos descargar y usar con Java Studio Creator 2. En las siguientes dos secciones demostraremos los componentes **AutoComplete Text Field** y **Map Viewer**.

Componente	Descripción
AutoComplete Text Field	Hace peticiones Ajax para mostrar una lista de sugerencias, a medida que el usuario escribe en el campo de texto.
Buy Now Button	Inicia una transacción a través del sitio Web PayPal.
Map Viewer	Usa la API de Google Maps para mostrar un mapa que permite inclinaciones, acercamientos y puede mostrar marcadores para las ubicaciones de interés.
Popup Calendar	Proporciona un calendario que permite a un usuario desplazarse entre los meses y años. Llena un **Campo de texto** con una fecha con formato cuando el usuario selecciona un día.
Progress Bar	Muestra en forma visual el progreso de una operación que tarda cierto tiempo en ejecutarse. Usa un cálculo suministrado por el programador para determinar el porcentaje de progreso.
Rating	Proporciona una barra de calificación personalizable de cinco estrellas, que puede mostrar mensajes a medida que el usuario mueve el ratón sobre las calificaciones.
Rich Textarea Editor	Proporciona un área de texto editable, que permite al usuario aplicar formato al texto con fuentes, colores, hipervínculos y fondos.
Select Value Text Field	Muestra una lista de sugerencias en una lista desplegable a medida que el usuario escribe, de manera similar al componente **AutoComplete Text Field**.

Figura 27.9 | Componentes habilitados para Ajax, proporcionados por la biblioteca de componentes BluePrints de Ajax.

Para utilizar los componentes Java BluePrints habilitados para Ajax en Java Studio Creator 2, debemos descargarlos e importarlos. El IDE proporciona un asistente para instalar este grupo de componentes. Para usarlo, seleccione **Herramientas > Centro de actualización** para mostrar el cuadro de diálogo **Asistente del centro de actualización**. Haga clic en **Siguiente >** para buscar actualizaciones disponibles. En el área **Nuevos módulos y actualizaciones disponibles** del cuadro de diálogo, seleccione **BluePrints AJAX Components** y haga clic con el botón derecho del ratón en el botón de flecha (**>**) para agregarlo a la lista de elementos que desea instalar. Haga clic en **Siguiente >** y siga los indicadores para aceptar las condiciones de uso y descargar los componentes. Cuando se complete la descarga, haga clic en **Siguiente >** y luego en **Terminar**. Haga clic en **Aceptar** para reiniciar el IDE.

A continuación, debe importar los componentes en la **Paleta**. Seleccione **Herramientas > Administrador de bibliotecas de componentes** y después haga clic en **Importar...**. Haga clic en el botón **Examinar...** en el cuadro de diálogo **Importar biblioteca de componentes** que aparezca. Seleccione el archivo `ui.complib` y haga clic en **Abrir**. Haga clic en **Aceptar** para importar los componentes **BluePrints AJAX Components** y **BluePrints AJAX SupportBeans**. Cierre el **Administrador de bibliotecas de componentes** para regresar al IDE.

Ahora deberá ver dos nuevos nodos en la parte inferior de la **Paleta**. El primero, **BluePrints AJAX Components**, proporciona los ocho componentes que se enlistan en la figura 27.9. El segundo, **BluePrints AJAX Support Beans**, incluye componentes que ofrecen soporte a los componentes Ajax. Ahora puede crear aplicaciones Web Ajax de alto rendimiento con sólo arrastrar, soltar y configurar las propiedades de los componentes, de igual forma que con los demás componentes en la **Paleta**.

27.4 AutoComplete Text Field y formularios virtuales

Vamos a demostrar el componente **AutoComplete Text Field** del catálogo BluePrints; para ello, hay que agregar un nuevo formulario a nuestra aplicación `LibretaDirecciones`. El componente **AutoComplete Text Field** proporciona una lista de sugerencias a medida que el usuario escribe. Obtiene las sugerencias de un origen de datos, que puede ser una base de datos o un servicio Web. En un momento dado, el nuevo formulario permitirá a los usuarios buscar en la libreta de direcciones por apellido paterno, y después por primer nombre. Si el usuario selecciona un contacto, la aplicación mostrará el nombre y la dirección de ese contacto en un mapa del vecindario. Vamos a crear este formulario en dos etapas. Primero, agregaremos el componente **AutoComplete Text Field** que mostrará las sugerencias a medida que el usuario escriba el apellido paterno de un contacto. Después agregaremos la funcionalidad de búsqueda y, en el tercer paso, la visualización de un mapa.

Agregar componentes de búsqueda a la página `LibretaDirecciones.jsp`

Utilice la aplicación `LibretaDirecciones` de la sección 27.2; suelte un componente **Texto estático** llamado `encabezadoBusqueda` debajo de `direccionesTabla`. Cambie su texto a `"Buscar en la libreta de direcciones por apellido:"` y cambie el tamaño de su fuente a `18 px`. Ahora arrastre un componente **AutoComplete Text Field** a la página y nómbrelo `nombreAutoComplete`. Establezca la propiedad `required` de este campo en `true`. Agregue una **Etiqueta** llamada `buscarNombreEtiqueta` que contenga el texto `"ApellidoPaterno:"` a la izquierda del componente **AutoComplete Text Field**. Por último, agregue un botón llamado `buscarBoton` con el texto `Buscar` a la derecha del componente **AutoComplete Text Field**.

27.4.1 Configuración de los formularios virtuales

Los formularios virtuales se utilizan cuando deseamos que un botón envíe un subconjunto de los campos de entrada de la página al servidor. Recuerde que los formularios virtuales internos de **Tabla** estaban habilitados, para que al hacer clic en los botones de paginación no se enviaran los datos de los componentes **Campo de texto** utilizados para agregar un contacto a la base de datos `LibretaDirecciones`. Los formularios virtuales son especialmente útiles para mostrar varios formularios en la misma página. Nos permiten especificar un **emisor** y uno o más **participantes** para un formulario. Cuando se hace clic en el componente emisor del formulario virtual, sólo se enviarán al servidor los valores de sus componentes participantes. Utilizamos formularios virtuales en nuestra aplicación `LibretaDirecciones` para separar el formulario para agregar un contacto a la base de datos `LibretaDirecciones` del formulario para buscar en la base de datos.

Para agregar formularios virtuales a la página, haga clic con el botón derecho en el botón **Enviar** que se encuentra en el formulario superior, y seleccione **Configurar formularios virtuales...** en el menú contextual para que aparezca el cuadro de diálogo **Configurar formularios virtuales**. Haga clic en **Nuevo** para agregar un formulario virtual; después haga clic en la columna **Nombre** y cambie el nombre del nuevo formulario a `agregarForm`. Haga doble clic en la columna **Enviar** y cambie la opción a **Sí** para indicar que este botón se debe utilizar para

enviar el formulario virtual `agregarForm`. Haga clic en **Aceptar** para salir del cuadro de diálogo. Después, seleccione todos los componentes **Campo de texto** utilizados para introducir la información de un contacto en el formulario superior. Para ello, mantenga oprimida la tecla *ctrl.* Mientras hace clic en cada **Campo de texto.** Haga clic con el botón derecho del ratón en uno de los componentes **Campo de texto** seleccionados y elija la opción **Configurar formularios virtuales….** En la columna **Función** del formulario `agregarForm`, cambie la opción a **Sí** para indicar que los valores en estos componentes **Campo de texto** deben enviarse al servidor cuando se envíe el formulario. En la figura 27.10 se muestra el cuadro de diálogo **Configurar formularios virtuales**, después de haber agregado ambos formularios virtuales. Haga clic en **Aceptar** para salir.

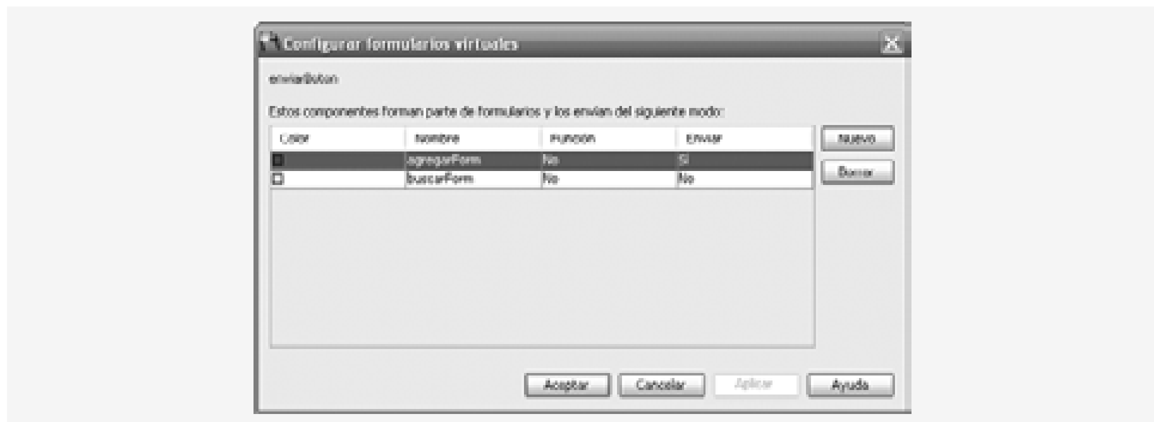


Figura 27.10 | Cuadro de diálogo **Configurar formularios virtuales**.

Repita el proceso antes descrito para crear un segundo formulario virtual llamado `buscarForm` para el formulario inferior. El botón **Buscar** deberá enviar el formulario `buscarForm`, y `nombreAutoComplete` deberá participar en este formulario. Después, regrese al modo **Diseño** y haga clic en el botón **Mostrar formularios virtuales** ( ) en la parte superior del panel Diseñador visual para mostrar una leyenda de los formularios virtuales en la página. Sus formularios virtuales deberán estar configurados como en la figura 27.11. Los componentes **Campo**



Figura 27.11 | Leyenda para los formularios virtuales.

de texto con el contorno de color azul participan en el formulario virtual `agregarForm`. Los componentes con el contorno de color verde participan en el formulario virtual `buscarForm`. Los componentes con el contorno de línea punteada envían sus respectivos formularios. Se proporciona una clave de colores en la parte inferior derecha del área de **Diseño**, para que usted sepa cuáles componentes pertenecen a cada formulario virtual.

27.4.2 Archivos JSP con formularios virtuales y un AutoComplete Text Field

La figura 27.12 presenta el archivo JSP generado por Java Studio Creator 2 para esta etapa de la aplicación `LibretaDirecciones`. Observe que se especifica una nueva biblioteca de etiquetas en el elemento raíz (`xmlns:bp="http://java.sun.com/blueprints/ui/14";` línea 6). Ésta es la biblioteca del catálogo de BluePrints que proporciona los componentes habilitados para Ajax, como el componente **AutoComplete Text Field**. Sólo nos enfocaremos en las nuevas características de esta JSP.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2
3  <!-- Fig. 27.12: LibretaDirecciones.jsp -->
4  <!-- JSP de LibretaDirecciones con un componente AutoComplete Text Field -->
5
6  <jsp:root version="1.2" xmlns:bp="http://java.sun.com/blueprints/ui/14"
7     xmlns:f="http://java.sun.com/jsf/core" xmlns:h=
8     "http://java.sun.com/jsf/html" xmlns:jsp="http://java.sun.com/JSP/Page"
9     xmlns:ui="http://www.sun.com/web/ui">
10    <jsp:directive.page contentType="text/html;charset=UTF-8"
11       pageEncoding="UTF-8"/>
12    <f:view>
13       <ui:page binding="#{LibretaDirecciones.page1}" id="page1">
14          <ui:html binding="#{LibretaDirecciones.html1}" id="html1">
15             <ui:head binding="#{LibretaDirecciones.head1}" id="head1">
16                <ui:link binding="#{LibretaDirecciones.link1}" id="link1"
17                   url="/resources/stylesheet.css"/>
18             </ui:head>
19             <ui:body binding="#{LibretaDirecciones.body1}" id="body1"
20                style="-rave-layout: grid">
21                <ui:form binding="#{LibretaDirecciones.form1}" id="form1"
22                   virtualFormsConfig="agregarForm | apaternoCampoTexto
23                   pnombreCampoTexto calleCampoTexto estadoCampoTexto
24                   cpCampoTexto ciudadCampoTexto | enviarBoton , buscarForm
25                   | nombreAutoComplete | buscarBoton">
26                   <ui:staticText binding="#{LibretaDirecciones.staticText1}" id=
27                      "staticText1" style="font-size: 18px; left: 12px;
28                      top: 24px; position: absolute" text=
29                      "Agregar un contacto a la libreta de direcciones:"/>
30                   <ui:textField binding="#{LibretaDirecciones.pnombreCampoTexto}"
31                      id="pnombreCampoTexto" maxLength="20" required="true"
32                      style="left: 132px; top: 72px;
33                      position: absolute"/>
34                   <ui:textField binding="#{LibretaDirecciones.apaternoCampoTexto}"
35                      id="apaternoCampoTexto" maxLength="30" required="true"
36                      style="left: 504px; top: 72px; position: absolute;
37                      width: 228px"/>
38                   <ui:textField binding="#{LibretaDirecciones.calleCampoTexto}"
39                      id="calleCampoTexto" maxLength="100" required="true"
40                      style="left: 132px; top: 96px; position: absolute;
41                      width: 600px"/>
42                   <ui:textField binding="#{LibretaDirecciones.ciudadCampoTexto}"
43                      id="ciudadCampoTexto" maxLength="30" required="true"
44                      style="left: 132px; top: 120px; position: absolute; width: 264px"/>
```

Figura 27.12 | JSP de `LibretaDirecciones` con un componente `AutoComplete TextField`. (Parte 1 de 4).

```
45                  <ui:textField binding="#{LibretaDirecciones.estadoCampoTexto}"
46                     id="estadoCampoTexto" maxLength="2" required="true"
47                     style="left: 480px; top: 120px; position: absolute;
48                     width: 60px"/>
49                  <ui:textField binding="#{LibretaDirecciones.cpCampoTexto}"
50                     id="cpCampoTexto" maxLength="5" required="true"
51                     style="left: 672px; top: 120px; position: absolute;
52                     width: 60px"/>
53                  <ui:label binding="#{LibretaDirecciones.pnombreEtiqueta}" for=
54                     "pnombreCampoTexto" id="pnombreEtiqueta" style="left: 12px;
55                     top: 72px; position: absolute" text="Primer nombre:"/>
56                  <ui:label binding="#{LibretaDirecciones.apaternoEtiqueta}" for=
57                     "apaternoCampoTexto" id="apaternoEtiqueta" style="position:
58                      absolute; left: 384px; top: 72px" text="Apellido Paterno:"/>
59                  <ui:label binding="#{LibretaDirecciones.calleEtiqueta}" for=
60                     "calleCampoTexto" id="calleEtiqueta" style="left: 12px;
61                      top: 96px; position: absolute" text="Calle:"/>
62                  <ui:label binding="#{LibretaDirecciones.ciudadEtiqueta}" for=
63                     "ciudadCampoTexto" id="ciudadEtiqueta" style="left: 12px;
64                      top: 120px; position: absolute" text="Ciudad:"/>
65                  <ui:label binding="#{LibretaDirecciones.estadoEtiqueta}" for=
66                     "estadoCampoTexto" id="estadoEtiqueta" style="left: 408px;
67                     top: 120px; position: absolute" text="Estado:"/>
68                  <ui:label binding="#{LibretaDirecciones.cpEtiqueta}" for=
69                     "cpCampoTexto" id="cpEtiqueta" style="height: 22px; left: 552px;
70                     top: 120px; position: absolute; width: 94px" text="Código postal:"/>
71                  <ui:button action="#{LibretaDirecciones.enviarBoton_action}"
72                     binding="#{LibretaDirecciones.enviarBoton}" id=
73                     "enviarBoton" primary="true" style="left: 131px;
74                     top: 168px; position: absolute" text="Enviar"/>
75                  <ui:button binding="#{LibretaDirecciones.borrarBoton}" id=
76                     "borrarBoton" reset="true" style="left: 251px; top: 168px;
77                     position: absolute" text="Borrar"/>
78                  <ui:table augmentTitle="false" binding=
79                     "#{LibretaDirecciones.direccionesTabla}" id="direccionesTabla"
80                     paginationControls="true" style="height: 56px;
81                     left: 12px; top: 204px; position: absolute; width: 720px"
82                     title="Contactos" width="720">
83                  <script><![CDATA[
84  <!--Las líneas 84 a 145 contienen código de JavaScript que se eliminó para ahorrar espacio.
85     El código fuente completo se proporciona en la carpeta de este ejemplo. -->
146                 }]]></script>
147                    <ui:tableRowGroup binding=
148                       "#{LibretaDirecciones.tableRowGroup1}"
149                       id="tableRowGroup1" rows="5" sourceData=
150                       "#{LibretaDirecciones.addressesDataProvider}"
151                       sourceVar="currentRow">
152                       <ui:tableColumn binding=
153                          "#{LibretaDirecciones.pnombreColumna}" headerText=
154                          "Primer nombre" id="pnombreColumna"
155                          sort="ADDRESSES.FIRSTNAME">
156                          <ui:staticText binding=
157                              "#{LibretaDirecciones.staticText2}" id=
158                              "staticText2" text="#{currentRow.value[
159                              'ADDRESSES.FIRSTNAME']}"/>
160                       </ui:tableColumn>
161                       <ui:tableColumn binding=
162                          "#{LibretaDirecciones.apaternoColumna}" headerText=
163                          "Apellido paterno" id="apaternoColumna"
```

Figura 27.12 | JSP de `LibretaDirecciones` con un componente `AutoComplete TextField`. (Parte 2 de 4).

```
         sort="ADDRESSES.LASTNAME">
         <ui:staticText binding=
            "#{LibretaDirecciones.staticText3}" id=
            "staticText3" text="#{currentRow.value[
            'ADDRESSES.LASTNAME']}"/>
      </ui:tableColumn>
      <ui:tableColumn binding=
         "#{LibretaDirecciones.calleColumna}" headerText=
         "Calle" id="calleColumna"
         sort="ADDRESSES.STREET">
         <ui:staticText binding=
            "#{LibretaDirecciones.staticText4}" id=
            "staticText4" text="#{currentRow.value[
            'ADDRESSES.STREET']}"/>
      </ui:tableColumn>
      <ui:tableColumn binding=
         "#{LibretaDirecciones.ciudadColumna}" headerText="Ciudad"
         id="ciudadColumna" sort="ADDRESSES.CITY">
         <ui:staticText binding=
            "#{LibretaDirecciones.staticText5}" id="staticText5"
            text="#{currentRow.value[
            'ADDRESSES.CITY']}"/>
      </ui:tableColumn>
      <ui:tableColumn binding=
         "#{LibretaDirecciones.estadoColumna}" headerText="Estado"
           id="estadoColumna" sort="ADDRESSES.STATE">
         <ui:staticText binding=
            "#{LibretaDirecciones.staticText6}" id=
            "staticText6" text="#{currentRow.value[
            'ADDRESSES.STATE']}"/>
      </ui:tableColumn>
      <ui:tableColumn binding="#{LibretaDirecciones.cpColumna}"
         headerText="CP" id="cpColumna"
         sort="ADDRESSES.ZIP">
         <ui:staticText binding=
            "#{LibretaDirecciones.staticText7}" id="staticText7"
            text="#{currentRow.value[
            'ADDRESSES.ZIP']}"/>
      </ui:tableColumn>
   </ui:tableRowGroup>
</ui:table>
<ui:messageGroup binding="#{LibretaDirecciones.messageGroup1}"
   id="messageGroup1" showGlobalOnly="true" style="height: 60px;
   left: 456px; top: 432px; position: absolute; width: 190px"/>
<ui:staticText binding="#{LibretaDirecciones.encabezadoBusqueda}"
   id="encabezadoBusqueda" style="font-size: 18px; left: 24px;
   top: 432px; position: absolute"
   text="Buscar en la libreta de direcciones por apellido:"/>
<ui:label binding="#{LibretaDirecciones.buscarNombreEtiqueta}"
   id="buscarNombreEtiqueta"
   style="left: 24px; top: 480px;
   position: absolute"
   text="Apellido paterno:"/>
<ui:label binding="#{LibretaDirecciones.buscarNombreEtiqueta}"
   for="nombreAutoComplete" id="buscarNombreEtiqueta"
   requiredIndicator="true"
   style="left: 24px; top: 480px; position: absolute"
   text="Apellido paterno:"/>
```

Figura 27.12 | JSP de `LibretaDirecciones` con un componente `AutoComplete TextField`. (Parte 3 de 4).

```
222                     <ui:button binding="#{LibretaDirecciones.buscarBoton}"
223                        id="buscarBoton" style="left: 359px; top: 480px;
224                        position: absolute" text="Buscar"/>
225                  </ui:form>
226               </ui:body>
227            </ui:html>
228         </ui:page>
229      </f:view>
230   </jsp:root>
```

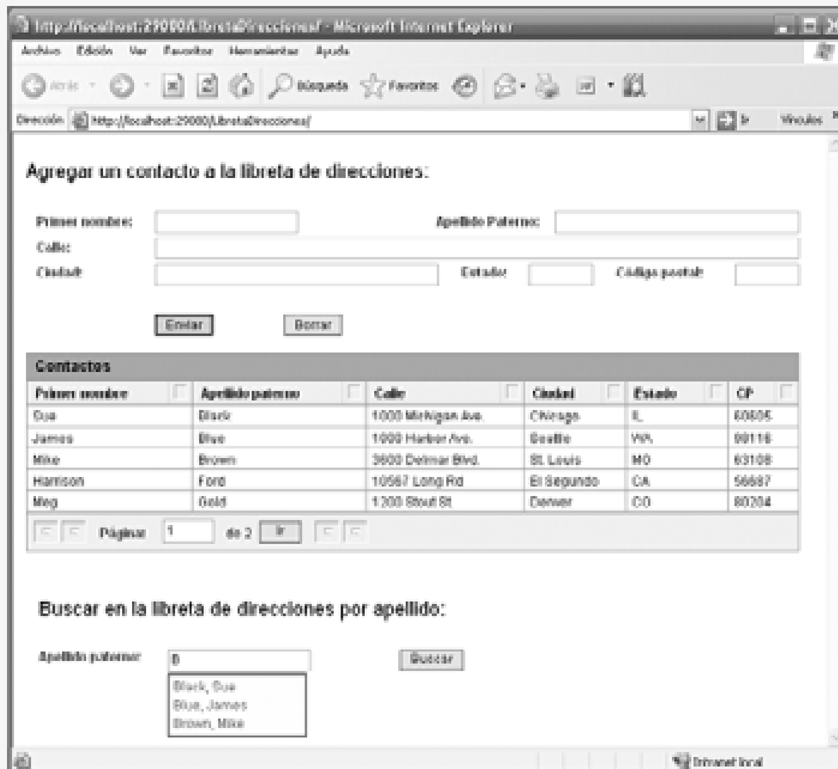


Figura 27.12 | JSP de `LibretaDirecciones` con un componente `AutoComplete TextField`. (Parte 4 de 4).

En las líneas 21 a 25 se configuran los formularios virtuales para esta página. En las líneas 217 a 221 se define el componente **AutoComplete Text Field**. El atributo `completionMethod` de este componente está enlazado al método `nombreAutoComplete_complete` del bean de página (que veremos en la sección 27.4.3), el cual proporciona la lista de opciones que debe sugerir el componente **AutoComplete Text Field**. Para crear este método, haga clic con el botón derecho en el componente `nombreAutoComplete` en vista de **Diseño** y seleccione **Editar controlador de eventos > complete**. Observe que el botón **Buscar** (líneas 222 a 224) no especifica un enlace con el método manejador de acciones; agregaremos esto en la sección 27.5.

27.4.3 Cómo proporcionar sugerencias para un AutoComplete Text Field

En la figura 27.13 se muestra el archivo de bean de página para la JSP de la figura 27.12. Incluye el método `nombreAutoComplete_complete`, el cual proporciona la funcionalidad para el componente **AutoComplete Text Field**. Aparte de este método, este bean de página es idéntico al de la figura 27.8.

```
// Fig. 27.13: LibretaDirecciones.java
// Bean de página para sugerir nombres en el componente AutoComplete Text Field.
package libretadirecciones;

import com.sun.data.provider.RowKey;
import com.sun.rave.web.ui.appbase.AbstractPageBean;
import com.sun.rave.web.ui.component.Body;
import com.sun.rave.web.ui.component.Form;
import com.sun.rave.web.ui.component.Head;
import com.sun.rave.web.ui.component.Html;
import com.sun.rave.web.ui.component.Link;
import com.sun.rave.web.ui.component.Page;
import javax.faces.FacesException;
import com.sun.rave.web.ui.component.StaticText;
import com.sun.rave.web.ui.component.TextField;
import com.sun.rave.web.ui.component.Label;
import com.sun.rave.web.ui.component.Button;
import com.sun.rave.web.ui.component.Table;
import com.sun.rave.web.ui.component.TableRowGroup;
import com.sun.rave.web.ui.component.TableColumn;
import com.sun.data.provider.impl.CachedRowSetDataProvider;
import com.sun.rave.web.ui.component.MessageGroup;
import com.sun.j2ee.blueprints.ui.autocomplete.AutoCompleteComponent;
import com.sun.j2ee.blueprints.ui.autocomplete.CompletionResult;
import javax.faces.context.FacesContext;

public class LibretaDirecciones extends AbstractPageBean
{
   private int __placeholder;

   private void _init() throws Exception
   {
      addressesDataProvider.setCachedRowSet(
         (javax.sql.rowset.CachedRowSet)
            getValue("#{SessionBean1.addressesRowSet}" ));
      direccionesTabla.setInternalVirtualForm(true);
   }

   // Las líneas 39 a 572 del código generado en forma automática se eliminaron para ahorrar espacio.
   // El código fuente completo se proporciona en la carpeta de este ejemplo.

   public void prerender()
   {
      addressesDataProvider.refresh();
   } // fin del método prerender

   public void destroy()
   {
      addressesDataProvider.close();
   } // fin del método destroy

   // manejador de acciones que agrega un contacto a la base de datos LibretaDirecciones
   // cuando el usuario hace clic en el botón Enviar
   public String enviarBoton_action()
   {
       if ( addressesDataProvider.canAppendRow() )
      {
         try
```

Figura 27.13 | Bean de página para sugerir nombres en el componente **AutoComplete Text Field**. (Parte 1 de 3).

```
            {
              RowKey rk = addressesDataProvider.appendRow();
              addressesDataProvider.setCursorRow(rk);

              addressesDataProvider.setValue( "ADDRESSES.FIRSTNAME",
                 pnombreCampoTexto.getValue() );
              addressesDataProvider.setValue( "ADDRESSES.LASTNAME",
                 apaternoCampoTexto.getValue() );
              addressesDataProvider.setValue( "ADDRESSES.STREET",
                 calleCampoTexto.getValue() );
              addressesDataProvider.setValue( "ADDRESSES.CITY",
                 ciudadCampoTexto.getValue() );
              addressesDataProvider.setValue( "ADDRESSES.STATE",
                 estadoCampoTexto.getValue() );
              addressesDataProvider.setValue( "ADDRESSES.ZIP",
                 cpCampoTexto.getValue());
              addressesDataProvider.commitChanges();

              // restablece los campos de texto
              apaternoCampoTexto.setValue( "" );
              pnombreCampoTexto.setValue( "" );
              calleCampoTexto.setValue( "" );
              ciudadCampoTexto.setValue( "" );
              estadoCampoTexto.setValue( "" );
              cpCampoTexto.setValue( "" );
            } // fin de try
            catch ( Exception ex )
            {
              error( "No se actualizo la libreta de direcciones." +
                 ex.getMessage() );
            } // fin de catch
         } // fin de if

         return null;
      } // fin del método enviarBoton_action


      // manejador de acciones para el cuadro autocompletar que obtiene los nombres
      // de la libreta de direcciones, cuyos prefijos coincidan con las letras escritas
      // hasta un momento dado, y los muestra en una lista de sugerencias.
      public void nombreAutoComplete_complete( FacesContext context, String
         prefix, CompletionResult result )
      {
         try
         {
            boolean tieneElSiguiente = addressesDataProvider.cursorFirst();

            while ( tieneElSiguiente )
            {
               // obtiene un nombre de la base de datos
               String nombre =
                  (String) addressesDataProvider.getValue(
                  "ADDRESSES.LASTNAME" ) + ", " +
                  (String) addressesDataProvider.getValue(
                  "ADDRESSES.FIRSTNAME" ) ;

               // si el nombre en la base de datos empieza con el prefijo, se
               // agrega a la lista de sugerencias
               if ( nombre.toLowerCase().startsWith( prefix.toLowerCase() ) )
```

Figura 27.13 | Bean de página para sugerir nombres en el componente **AutoComplete Text Field**. (Parte 2 de 3).

```
649                {
650                   result.addItem( nombre );
651                } // fin de if
652                else
653                {
654                   // termina el ciclo si el resto de los nombres son
655                   // alfabéticamente menores que el prefijo
656                   if ( prefix.compareTo( nombre ) < 0 )
657                   {
658                     break;
659                   } // fin de if
660                } // fin de else
661
662                // desplaza el cursor a la siguiente fila de la base de datos
663                tieneElSiguiente = addressesDataProvider.cursorNext();
664             } // fin de while
665          } // fin de try
666          catch ( Exception ex )
667          {
668             result.addItem( "Excepcion al obtener nombres que coincidan." );
669          } // fin de catch
670       } // fin del método nombreAutoComplete_complete
671    } // fin de la clase LibretaDirecciones
```

Figura 27.13 | Bean de página para sugerir nombres en el componente **AutoComplete Text Field**. (Parte 3 de 3).

El método `nombreAutoComplete_complete` (líneas 630 a 670) se invoca después de cada pulsación de tecla en el componente **AutoComplete Text Field**, para actualizar la lista de sugerencias con base en el texto que el usuario ha escrito hasta cierto punto. El método recibe una cadena (`prefix`) que contiene el texto que introdujo el usuario, y un objeto `CompletionResult` (`result`) que se utiliza para mostrar sugerencias al usuario. El método itera a través de las filas del objeto `addressesDataProvider`, obtiene el nombre de cada fila, comprueba si el nombre empieza con las letras escritas hasta cierto punto y, de ser así, agrega el nombre a `result`. En la línea 635 se establece el cursor a la primera fila en el proveedor de datos. En la línea 637 se determina si hay más filas en el proveedor de datos. De ser así, en las líneas 640 a 644 se obtienen el apellido paterno y el primer nombre de la fila actual, y se crea un objeto `String` en el formato *apellido paterno*, *primer nombre*. En la línea 648 se comparan las versiones en minúscula de `nombre` y `prefix` para determinar si el nombre empieza con los caracteres escritos hasta ahora. De ser así, el nombre es una coincidencia y en la línea 650 se agrega a `result`.

Recuerde que el proveedor de datos envuelve un objeto `CachedRowSet` que contiene una consulta SQL que devuelve las filas en la base de datos ordenada por apellido paterno, y después por primer nombre. Esto nos permite iterar a través del proveedor de datos, una vez que llegamos a una fila cuyo nombre va alfabéticamente después que el texto introducido por el usuario; los nombres en las filas más allá de esto serán alfabéticamente mayores y, por ende, no son coincidencias potenciales. Si el `nombre` no coincide con el texto introducido hasta un momento dado, en la línea 656 se evalúa si el nombre actual es alfabéticamente mayor que el prefijo (`prefix`). De ser así, en la línea 658 se termina el ciclo.



Tip de rendimiento 27.1

Al usar columnas de la base de datos para proporcionar sugerencias en un componente **AutoComplete Text Field***, si ordenamos las columnas eliminamos la necesidad de comprobar cada fila en la base de datos, en búsqueda de coincidencias potenciales. Esto mejora considerablemente el rendimiento cuando se maneja una base de datos extensa.*

Si el nombre no es una coincidencia, ni es alfabéticamente mayor que `prefix`, entonces en la línea 663 se desplaza el cursor a la siguiente fila en el proveedor de datos. Si hay otra fila, el ciclo vuelve a iterar, comprobando si el nombre en la siguiente fila coincide con el valor de `prefix` y debe agregarse a `results`.

En las líneas 666 a 669 se atrapan las excepciones que se generen mientras se realiza la búsqueda en la base de datos. En la línea 668 se agrega texto al cuadro de sugerencias, indicando el error al usuario.

27.5 Componente Map Viewer de Google Maps

Ahora completaremos la aplicación LibretaDirecciones, para lo cual agregaremos funcionalidad al **Botón Buscar**. Cuando el usuario hace clic en este **Botón**, el nombre en el componente **AutoComplete Text Field** se utiliza para buscar en la base de datos LibretaDirecciones. También agregamos a la página un componente JSF **Map Viewer** habilitado para Ajax, para mostrar un mapa del área para esa dirección. Un componente **Map Viewer** utiliza el servicio Web de la **API de Google Maps** para buscar y mostrar mapas. (En el capítulo 28 hablaremos sobre los detalles de los servicios Web). En este ejemplo, utilizar la API de Google Maps es un proceso análogo a crear llamadas a métodos ordinarios en un objeto **Map Viewer** y su bean de soporte en el archivo de bean de página. Al encontrar un contacto, mostramos un mapa del vecindario con un componente **Map Viewer** que apunta a la ubicación e indica el nombre del contacto y su dirección.

27.5.1 Cómo obtener una clave de la API Google Maps

Para utilizar el componente **Map Viewer**, debe tener una cuenta con Google. Visite el sitio https://www.google.com/accounts/ManageAccount para registrarse y obtener una cuenta gratuita, si no tiene una ya. Una vez que haya iniciado sesión en su cuenta, debe obtener una clave para usar la API de Google Maps en www.google.com/apis/maps. La clave que reciba será específica para esta aplicación Web y limitará el número de mapas que puede mostrar la aplicación por día. Cuando se registre para la clave, tendrá que escribir el URL para la aplicación que utilizará la API de Google Maps. Si va a desplegar la aplicación sólo en el servidor de prueba Sun Application Server 8 de Java Studio Creator 2, escriba http://localhost:29080 como el URL.

Una vez que acepte los términos y condiciones de Google, será redirigido a una página que contendrá su nueva clave para la API de Google Maps. Guarde esta clave en un archivo de texto, en una ubicación conveniente para una futura referencia.

27.5.2 Cómo agregar un componente y un Map Viewer a una página

Ahora que tiene una clave para usar la API de Google Maps, está listo para completar la aplicación LibretaDirecciones. Con el archivo LibretaDirecciones.jsp abierto en modo **Diseño**, agregue un componente **Map Viewer** llamado mapViewer debajo del componente nombreAutoComplete. En la ventana **Propiedades**, establezca la propiedad clave del componente **Map Viewer** con la clave que obtuvo para acceder a la API de Google Maps. Establezca la propiedad rendered en false, de manera que el mapa no se muestre cuando el usuario todavía no haya buscado una dirección. Establezca la propiedad zoomLevel en 1 (In), de manera que el usuario pueda ver los nombres de las calles en el mapa.

Suelte un componente **Map Marker** (llamado mapMarker) de la sección **AJAX Support Beans** de la **Paleta** en cualquier parte de la página. Este componente (que no está visible en modo **Diseño**) marca la ubicación del contacto en el mapa. Debe enlazar el marcador con el mapa, de manera que se muestre el marcador en el mapa. Para ello, haga clic con el botón derecho en el componente **Map Viewer** en modo **Diseño** y seleccione **Enlaces de propiedades...** para mostrar el cuadro de diálogo **Enlaces de propiedades**. Seleccione info de la columna **Seleccionar propiedad enlazable** del cuadro de diálogo, y después seleccione mapMarker de la columna **Seleccionar destino de enlace**. Haga clic en **Aplicar** y después en **Cerrar**.

Por último, suelte un componente **Geocoding Service Object** (llamado geoCoder) de la sección **AJAX Support Beans** de la **Paleta**, en cualquier parte de la página. Este objeto (que no está visible en modo **Diseño**) convierte las direcciones de las calles en latitudes y longitudes que el componente **Map Viewer** utiliza para mostrar un mapa apropiado.

Cómo agregar un proveedor de datos a la página

Para completar esta aplicación, necesita un segundo proveedor de datos para buscar en la base de datos LibretaDirecciones, con base en el primer nombre y apellido paterno introducidos en el componente **AutoComplete Text Field**. Abra la ventana **Servidores** y expanda el nodo **LibretaDirecciones** junto con su nodo **Tablas** para revelar la tabla **Addresses**. Haga clic con el botón derecho del ratón en el nodo de la tabla y seleccione **Agregar a página** para mostrar el cuadro de diálogo **Agregar proveedor de datos con RowSet** (figura 27.14). Queremos crear un nuevo origen de datos, en vez de utilizar el existente, ya que la consulta para buscar contactos es distinta de la consulta para mostrar todos los contactos. Seleccione la opción **Crear** para el objeto SessionBean1 y escriba el nombre busquedaDirecciones para el proveedor de datos. Haga clic en **Aceptar** para crear el nuevo proveedor de datos. En la ventana **Esquema**, se ha agregado un nuevo nodo llamado busquedaDireccionesDataProvider



Figura 27.14 | Cuadro de diálogo para crear un nuevo proveedor de datos.

al nodo `LibretaDirecciones`, y se ha agregado un nodo llamado `busquedaDirecciones` al nodo `SessionBean`.

Haga doble clic en el nodo `busquedaDirecciones` para editar la instrucción SQL para este objeto `RowSet`. Como vamos a usar este conjunto de filas para buscar en la base de datos un apellido paterno y un primer nombre, necesitamos agregar parámetros de búsqueda a la instrucción `SELECT` que ejecutará el objeto `RowSet`. Para ello, escriba el texto `"= ?"` en la columna **Criterios** de las filas del primer nombre y apellido paterno en la tabla del editor de instrucciones SQL. El número `1` deberá aparecer en la columna **Orden** para el primer nombre, y el número `2` para el apellido paterno. Observe que se han agregado las líneas

```
WHERE JHTP7.ADDRESSES.FIRSTNAME = ?
    AND JHTP7.ADDRESSES.LASTNAME = ?
```

a la instrucción SQL. Esto indica que el objeto `RowSet` ahora ejecuta una instrucción SQL con parámetros. Estos parámetros se pueden establecer mediante programación, en donde el primer nombre es el primer parámetro y el apellido paterno es el segundo.

27.5.3 Archivo JSP con un componente `Map Viewer`

La figura 27.15 presenta el archivo JSP para la aplicación de libreta de direcciones completa. Es casi idéntico al archivo JSP de las dos versiones anteriores de esta aplicación. La nueva característica es el componente **Map Viewer** (y sus componentes de soporte) que se utiliza para mostrar un mapa con la ubicación del contacto. Sólo hablaremos de los nuevos elementos de este archivo. [*Nota:* este código no se ejecutará sino hasta que haya especificado su propia clave Google Maps en las líneas 227 a 229. Puede pegar su clave en la propiedad **key** del componente **Map Viewer** en la ventana **Propiedades**].

```
<?xml version="1.0" encoding="UTF-8"?>

<!-- Fig. 27.15: LibretaDirecciones.jsp -->
<!-- Página JSP de LibretaDirecciones con un componente Map Viewer. -->

<jsp:root version="1.2" xmlns:bp="http://java.sun.com/blueprints/ui/14"
```

Figura 27.15 | JSP de `LibretaDirecciones` con un componente **Map Viewer**. (Parte 1 de 5).

```
    xmlns:f="http://java.sun.com/jsf/core" xmlns:h=
    "http://java.sun.com/jsf/html" xmlns:jsp="http://java.sun.com/JSP/Page"
    xmlns:ui="http://www.sun.com/web/ui">
    <jsp:directive.page contentType="text/html;charset=UTF-8"
       pageEncoding="UTF-8"/>
    <f:view>
       <ui:page binding="#{LibretaDirecciones.page1}" id="page1">
          <ui:html binding="#{LibretaDirecciones.html1}" id="html1">
           <ui:head binding="#{LibretaDirecciones.head1}" id="head1">
              <ui:link binding="#{LibretaDirecciones.link1}" id="link1"
                url="/resources/stylesheet.css"/>
            </ui:head>
            <ui:body binding="#{LibretaDirecciones.body1}" id="body1"
              style="-rave-layout: grid">
              <ui:form binding="#{LibretaDirecciones.form1}" id="form1"
                virtualFormsConfig="agregarForm | apaternoCampoTexto
                pnombreCampoTexto calleCampoTexto estadoCampoTexto
                cpCampoTexto ciudadCampoTexto | enviarBoton , buscarForm
                | nombreAutoComplete | buscarBoton">
                <ui:staticText binding="#{LibretaDirecciones.staticText1}" id=
                    "staticText1" style="font-size: 18px; left: 12px;
                    top: 24px; position: absolute"
                    text="Agregar un contacto a la libreta de direcciones:"/>
                <ui:textField binding="#{LibretaDirecciones.pnombreCampoTexto}"
                    id="pnombreCampoTexto" maxLength="20" required="true"
                    style="left: 132px; top: 72px;
                    position: absolute"/>
                <ui:textField binding="#{LibretaDirecciones.apaternoCampoTexto}"
                    id="apaternoCampoTexto" maxLength="30" required="true"
                    style="left: 504px; top: 72px; position: absolute;
                     width: 228px"/>
                <ui:textField binding="#{LibretaDirecciones.calleCampoTexto}"
                    id="calleCampoTexto" maxLength="100" required="true"
                    style="left: 132px; top: 96px; position: absolute;
                     width: 600px"/>
                <ui:textField binding="#{LibretaDirecciones.ciudadCampoTexto}"
                    id="ciudadCampoTexto" maxLength="30" required="true"
                    style="left: 132px; top: 120px; position: absolute; width: 264px"/>
                <ui:textField binding="#{LibretaDirecciones.estadoCampoTexto}"
                    id="estadoCampoTexto" maxLength="2" required="true"
                    style="left: 480px; top: 120px; position: absolute;
                    width: 60px"/>
                <ui:textField binding="#{LibretaDirecciones.cpCampoTexto}"
                    id="cpCampoTexto" maxLength="5" required="true"
                    style="left: 672px; top: 120px; position: absolute;
                    width: 60px"/>
                <ui:label binding="#{LibretaDirecciones.pnombreEtiqueta}" for=
                    "pnombreCampoTexto" id="pnombreEtiqueta" style="left: 12px;
                    top: 72px; position: absolute" text="Primer nombre:"/>
                 <ui:label binding="#{LibretaDirecciones.apaternoEtiqueta}" for=
                    "apaternoCampoTexto" id="apaternoEtiqueta" style="position:
                    absolute; left: 384px; top: 72px" text="Apellido Paterno:"/>
                <ui:label binding="#{LibretaDirecciones.calleEtiqueta}" for=
                    "calleCampoTexto" id="calleEtiqueta" style="left: 12px;
                    top: 96px; position: absolute" text="Calle:"/>
                <ui:label binding="#{LibretaDirecciones.ciudadEtiqueta}" for=
                    "ciudadCampoTexto" id="ciudadEtiqueta" style="left: 12px;
                    top: 120px; position: absolute" text="Ciudad:"/>
                <ui:label binding="#{LibretaDirecciones.estadoEtiqueta}"
```

Figura 27.15 | JSP de `LibretaDirecciones` con un componente **Map Viewer**. (Parte 2 de 5).

```
                        for="estadoCampoTexto" id="estadoEtiqueta" style="left: 408px;
                        top: 120px; position: absolute" text="Estado:"/>
                    <ui:label binding="#{LibretaDirecciones.cpEtiqueta}" for=
                        "cpCampoTexto" id="cpEtiqueta" style="height: 22px; left: 552px;
                        top: 120px; position: absolute; width: 94px" text="Código
                         postal:"/>
                    <ui:button action="#{LibretaDirecciones.enviarBoton_action}"
                        binding="#{LibretaDirecciones.enviarBoton}" id=
                         "enviarBoton" primary="true" style="left: 131px;
                        top: 168px; position: absolute" text="Enviar"/>
                    <ui:button binding="#{LibretaDirecciones.borrarBoton}" id=
                        "borrarBoton" reset="true" style="left: 251px; top: 168px;
                        position: absolute" text="Borrar"/>
                    <ui:table augmentTitle="false" binding=
                        "#{LibretaDirecciones.direccionesTabla}" id="direccionesTabla"
                        paginationControls="true" style="height: 56px; left: 12px;
                        top: 204px; position: absolute; width: 720px"
                        title="Contactos" width="720">
                     <script><![CDATA[
<!--Las líneas 84 a 145 contienen código de JavaScript que se eliminó para ahorrar
   espacio.
   El código fuente completo se proporciona en la carpeta de este ejemplo. -->
                     }]]></script>
                        <ui:tableRowGroup binding=
                          "#{LibretaDirecciones.tableRowGroup1}" id=
                          "tableRowGroup1" rows="5" sourceData=
                          "#{LibretaDirecciones.addressesDataProvider}"
                          sourceVar="currentRow">
                          <ui:tableColumn binding=
                             "#{LibretaDirecciones.pnombreColumna}" headerText=
                             "Primer nombre" id="pnombreColumna"
                            sort="ADDRESSES.FIRSTNAME">
                            <ui:staticText binding=
                              "#{LibretaDirecciones.staticText2}"
                              id="staticText2" text="#{currentRow.value[
                              'ADDRESSES.FIRSTNAME']}"/>
                          </ui:tableColumn>
                          <ui:tableColumn binding=
                            "#{LibretaDirecciones.apaternoColumna}" headerText=
                            "Apellido paterno" id="apaternoColumna"
                            sort="ADDRESSES.LASTNAME">
                             <ui:staticText binding=
                              "#{LibretaDirecciones.staticText3}" id=
                              "staticText3" text="#{currentRow.value[
                              'ADDRESSES.LASTNAME']}"/>
                          </ui:tableColumn>
                          <ui:tableColumn binding=
                            "#{LibretaDirecciones.calleColumna}" headerText=
                             "Calle" id="calleColumna"
                            sort="ADDRESSES.STREET">
                            <ui:staticText binding=
                               "#{LibretaDirecciones.staticText4}" id=
                               "staticText4" text="#{currentRow.value[
                               'ADDRESSES.STREET']}"/>
                          </ui:tableColumn>
                          <ui:tableColumn binding=
                            "#{LibretaDirecciones.ciudadColumna}" headerText=
```

Figura 27.15 | JSP de `LibretaDirecciones` con un componente **Map Viewer**. (Parte 3 de 5).

```
                              "Ciudad" id="ciudadColumna" sort="ADDRESSES.CITY">
                               <ui:staticText binding=
                                 "#{LibretaDirecciones.staticText5}" id="staticText5"
                                 text="#{currentRow.value[
                                 'ADDRESSES.CITY']}"/>
                           </ui:tableColumn>
                           <ui:tableColumn binding=
                              "#{LibretaDirecciones.estadoColumna}" headerText="Estado"
                              id="estadoColumna" sort="ADDRESSES.STATE">
                              <ui:staticText binding=
                                 "#{LibretaDirecciones.staticText6}" id=
                                 "staticText6" text="#{currentRow.value[
                                 'ADDRESSES.STATE']}"/>
                           </ui:tableColumn>
                           <ui:tableColumn binding="#{LibretaDirecciones.cpColumna}"
                              headerText="CP" id="cpColumna"
                              sort="ADDRESSES.ZIP">
                              <ui:staticText binding=
                                 "#{LibretaDirecciones.staticText7}" id="staticText7"
                                  text="#{currentRow.value[
                                 'ADDRESSES.ZIP']}"/>
                           </ui:tableColumn>
                        </ui:tableRowGroup>
                    </ui:table>
                    <ui:messageGroup binding="#{LibretaDirecciones.messageGroup1}"
                        id="messageGroup1" showGlobalOnly="true" style="height: 60px;
                        left: 456px; top: 408px; position: absolute; width: 190px"/>
                    <ui:staticText binding="#{LibretaDirecciones.encabezadoBusqueda}" id=
                        "encabezadoBusqueda" style="font-size: 18px; left: 24px;
                        top: 432px; position: absolute"
                        text="Buscar en la libreta de direcciones por apellido:"/>
                    <ui:label binding="#{LibretaDirecciones.buscarNombreEtiqueta}" for=
                        "nombreAutoComplete" id="buscarNombreEtiqueta"
                        requiredIndicator="true" style="left: 24px; top: 480px;
                        position: absolute" text="Apellido paterno:"/>
                    <bp:autoComplete binding=
                        "#{LibretaDirecciones.nombreAutoComplete}" completionMethod=
                        "#{LibretaDirecciones.nombreAutoComplete_complete}"
                        id="nombreAutoComplete" required="true" style="left:
                        144px; top: 480px; position: absolute"/>
                    <ui:button action="#{LibretaDirecciones.buscarBoton_action}"
                        binding="#{LibretaDirecciones.buscarBoton}" id=
                        "buscarBoton" style="left: 359px; top: 480px; position:
                         absolute" text="Buscar"/>
                    <bp:mapViewer binding="#{LibretaDirecciones.mapViewer}" center=
                        "#{LibretaDirecciones.mapViewer_center}" id="mapViewer"
                        info="#{LibretaDirecciones.mapMarker}" key=
                        "ABQIAAAAxDzuwvNQoM33k5O8FmOI9BRv3hXvfLy8rd_zkEeAYi6qFBadthS
                        as7kIyZ1EERRCUWTUqIrqGp8ybg" mapControls="false"
                        style="height: 550px; left: 24px; top: 528px; position:
                        absolute; width: 718px" zoomLevel="1"/>
                  </ui:form>
               </ui:body>
            </ui:html>
         </ui:page>
      </f:view>
   </jsp:root>
```

Figura 27.15 | JSP de `LibretaDirecciones` con un componente **Map Viewer**. (Parte 4 de 5).

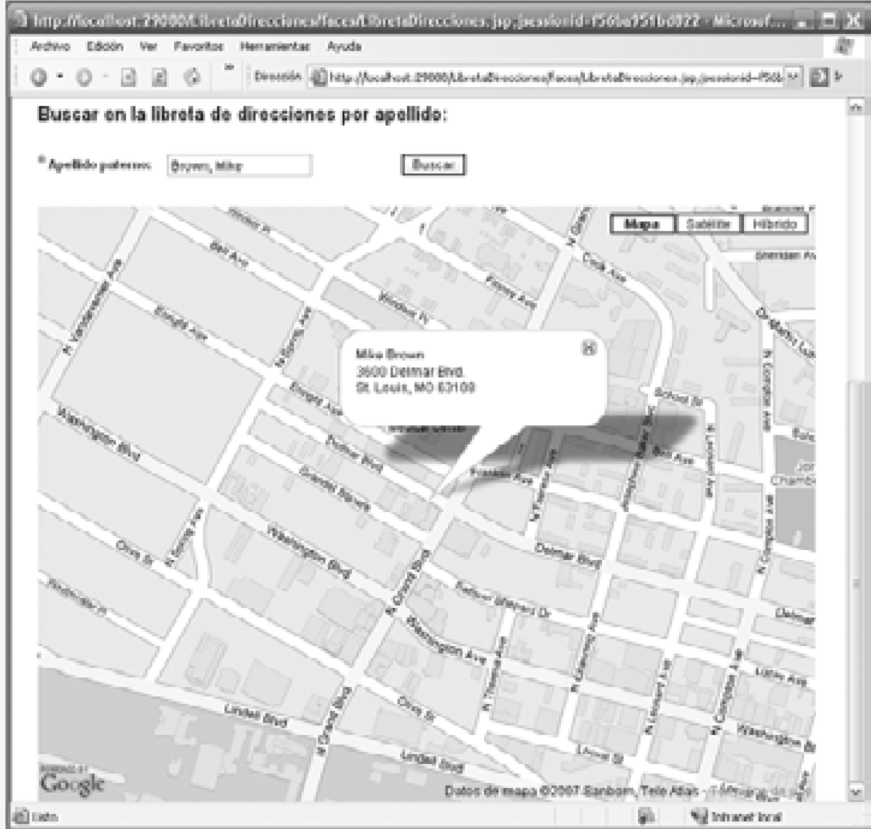


Figura 27.15 | JSP de `LibretaDirecciones` con un componente **Map Viewer**. (Parte 5 de 5).

En las líneas 242 a 247 se define el componente `mapViewer` que muestra un mapa del área que rodea a la dirección. El atributo `center` del componente está enlazado a la propiedad `mapViewer_center` del bean de página. Esta propiedad se manipula en el archivo de bean de página, para centrar el mapa en la dirección deseada.

El atributo `action` del `botón` **Buscar** ahora está enlazado al método `buscarBoton_action` en el bean de página (línea 226). Este manejador de acciones busca en la base de datos `LibretaDirecciones` el nombre introducido en el componente **AutoComplete Text Field** y muestra el nombre del contacto, junto con la dirección, en un mapa de la ubicación de ese contacto.

27.5.4 Bean de página que muestra un mapa en el componente Map Viewer

En la figura 27.16 se presenta el bean de página para la aplicación `LibretaDirecciones` completa. La mayor parte de este archivo es idéntico a los beans de página para las primeras dos versiones de esta aplicación. Hablaremos sólo del nuevo método manejador de acciones, `buscarBoton_action`.

El método `buscarBoton_action` (líneas 646 a 704) se invoca cuando el usuario hace clic en el botón **Buscar** en el formulario inferior de la página. Las líneas 649 a 652 recuperan el nombre del componente **AutoComplete Text Field** y lo separan en cadenas para el primer nombre y el apellido paterno. En cada una de las líneas 662 a 669 se obtiene el objeto `CachedRowSet` de `busquedaDireccionesDataProvider`, y después se utiliza su método `setObject` para establecer los parámetros de la consulta con el primer nombre y el apellido paterno. El método `setObject` sustituye un parámetro en la consulta SQL con una cadena especificada. En la línea 661 se actualiza el proveedor de datos, el cual ejecuta la consulta del objeto `RowSet` envuelto con los nuevos parámetros. El conjunto de resultados ahora contiene sólo filas que coinciden con el primer nombre y el apellido paterno del componente **AutoComplete Text Field.** En las líneas 662 a 669 se obtienen de la base de datos la dirección de

la calle, la ciudad, el estado y el código postal para este contacto. Observe que en este ejemplo, suponemos que no hay varias entradas en la libreta de direcciones para los mismos valores de primer nombre y apellido paterno, ya que sólo obtenemos la información de la dirección para la primera fila en el proveedor de datos. Cualquier fila adicional que coincida con el primer nombre y el apellido paterno se ignora.

```
// Fig. 27.16: LibretaDirecciones.java
// Bean de página para agregar un contacto a la liberta de direcciones.
package libretadirecciones;

import com.sun.data.provider.RowKey;
import com.sun.rave.web.ui.appbase.AbstractPageBean;
import com.sun.rave.web.ui.component.Body;
import com.sun.rave.web.ui.component.Form;
import com.sun.rave.web.ui.component.Head;
import com.sun.rave.web.ui.component.Html;
import com.sun.rave.web.ui.component.Link;
import com.sun.rave.web.ui.component.Page;
import javax.faces.FacesException;
import com.sun.rave.web.ui.component.StaticText;
import com.sun.rave.web.ui.component.TextField;
import com.sun.rave.web.ui.component.Label;
import com.sun.rave.web.ui.component.Button;
import com.sun.rave.web.ui.component.Table;
import com.sun.rave.web.ui.component.TableRowGroup;
import com.sun.rave.web.ui.component.TableColumn;
import com.sun.data.provider.impl.CachedRowSetDataProvider;
import com.sun.rave.web.ui.component.MessageGroup;
import com.sun.j2ee.blueprints.ui.autocomplete.AutoCompleteComponent;
import com.sun.j2ee.blueprints.ui.autocomplete.CompletionResult;
import javax.faces.context.FacesContext;
import com.sun.j2ee.blueprints.ui.mapviewer.MapComponent;
import com.sun.j2ee.blueprints.ui.mapviewer.MapPoint;
import com.sun.j2ee.blueprints.ui.geocoder.GeoCoder;
import com.sun.j2ee.blueprints.ui.geocoder.GeoPoint;
import com.sun.j2ee.blueprints.ui.mapviewer.MapMarker;

public class LibretaDirecciones extends AbstractPageBean
{
   private int __placeholder;

   private void _init() throws Exception
   {
      addressesDataProvider.setCachedRowSet(
         (javax.sql.rowset.CachedRowSet)
            getValue("#{SessionBean1.addressesRowSet}"));
      direccionesTabla.setInternalVirtualForm(true);
      busquedaDireccionesDataProvider.setCachedRowSet(
         (javax.sql.rowset.CachedRowSet)
            getValue("#{SessionBean1.busquedaDirecciones}"));
      mapViewer.setRendered(false);
   } // fin del método _init

   // Las líneas 48 a 544 del código generado en forma automática se eliminaron para ahorrar espacio.
   // El código fuente completo se proporciona en la carpeta de este ejemplo.

   public void prerender()
```

Figura 27.16 | Bean de página que obtiene un mapa para mostrarlo en el componente `MapViewer`. (Parte 1 de 4).

```
   {
      addressesDataProvider.refresh();
   } // fin del método prerender

   public void destroy()
   {
      busquedaDireccionesDataProvider.close();
      addressesDataProvider.close();
   } // fin del método destroy

   // manejador de acciones que agrega un contacto a la base de datos LibretaDirecciones
   // cuando el usuario hace clic en el botón Enviar
   public String enviarBoton_action()
   {
      if ( addressesDataProvider.canAppendRow() )
      {
         try
         {
            RowKey rk = addressesDataProvider.appendRow();
            addressesDataProvider.setCursorRow(rk);

            addressesDataProvider.setValue( "ADDRESSES.FIRSTNAME",
               pnombreCampoTexto.getValue() );
            addressesDataProvider.setValue( "ADDRESSES.LASTNAME",
               apaternoCampoTexto.getValue() );
            addressesDataProvider.setValue( "ADDRESSES.STREET",
               calleCampoTexto.getValue() );
            addressesDataProvider.setValue( "ADDRESSES.CITY",
               ciudadCampoTexto.getValue() );
            addressesDataProvider.setValue( "ADDRESSES.STATE",
               estadoCampoTexto.getValue() );
            addressesDataProvider.setValue( "ADDRESSES.ZIP",
               cpCampoTexto.getValue());
            addressesDataProvider.commitChanges();

            // restablece los campos de texto
            apaternoCampoTexto.setValue( "" );
            pnombreCampoTexto.setValue( "" );
            calleCampoTexto.setValue( "" );
            ciudadCampoTexto.setValue( "" );
            estadoCampoTexto.setValue( "" );
            cpCampoTexto.setValue( "" );
         } // fin de try
         catch ( Exception ex )
         {
            error( "No se actualizo la libreta de direcciones." +
               ex.getMessage() );
         } // fin de catch
      } // fin de if

      return null;
   } // fin del método enviarBoton_action

   // manejador de acciones para el cuadro autocompletar que obtiene los nombres
   // de la libreta de direcciones, cuyos prefijos coincidan con las letras escritas
   // hasta un momento dado, y los muestra en una lista de sugerencias.
   public void nombreAutoComplete_complete( FacesContext context, String
      prefix, CompletionResult result )
   {
```

Figura 27.16 | Bean de página que obtiene un mapa para mostrarlo en el componente MapViewer. (Parte 2 de 4).

```
      try
      {
         boolean tieneElSiguiente = addressesDataProvider.cursorFirst();

         while ( tieneElSiguiente )
         {
            // obtiene un nombre de la base de datos
            String nombre =
               (String) addressesDataProvider.getValue(
               "ADDRESSES.LASTNAME" ) + ", " +
               (String) addressesDataProvider.getValue(
               "ADDRESSES.FIRSTNAME" ) ;

            // si el nombre en la base de datos empieza con el prefijo, se
            // agrega a la lista de sugerencias
            if ( nombre.toLowerCase().startsWith( prefix.toLowerCase() ) )
            {
               result.addItem( nombre );
            } // fin de if
            else
            {
               // termina el ciclo si el resto de los nombres son
               // alfabéticamente menores que el prefijo
               if ( prefix.compareTo( nombre ) < 0 )
               {
                  break;
               } // fin de if
            } // fin de else

            // desplaza el cursor a la siguiente fila de la base de datos
            tieneElSiguiente = addressesDataProvider.cursorNext();
         } // fin de while
      } // fin de try
      catch ( Exception ex )
      {
         result.addItem( "Excepcion al obtener nombres que coincidan." );
      } // fin de catch
   } // fin del método nombreAutoComplete_complete

   // manejador de acciones para el buscarBoton que busca en la base de datos de
   // la libreta de direcciones y muestra la dirección solicitada en un mapa correspondiente.
   public String buscarBoton_action()
   {
      // divide el texto del campo AutoComplete en primer nombre y apellido
      String nombre = String.valueOf( nombreAutoComplete.getValue() );
      int splitIndex = nombre.indexOf( "," );
      String apaterno = nombre.substring( 0, splitIndex );
      String pnombre = nombre.substring( splitIndex + 2 );

      try
      {
         // establece los parámetros para la consulta direccionesSeleccionadas
         busquedaDireccionesDataProvider.getCachedRowSet().setObject(
            1, pnombre );
         busquedaDireccionesDataProvider.getCachedRowSet().setObject(
            2, apaterno );
         busquedaDireccionesDataProvider.refresh();
         String calle = (String) busquedaDireccionesDataProvider.getValue(
            "ADDRESSES.STREET" );
```

Figura 27.16 | Bean de página que obtiene un mapa para mostrarlo en el componente `MapViewer`. (Parte 3 de 4).

```
664            String ciudad = (String) busquedaDireccionesDataProvider.getValue(
665               "ADDRESSES.CITY" );
666            String estado = (String) busquedaDireccionesDataProvider.getValue(
667               "ADDRESSES.STATE" );
668            String cp =  (String) busquedaDireccionesDataProvider.getValue(
669               "ADDRESSES.ZIP" );
670
671            // aplica formato a la dirección para Google Maps
672            String direccionGoogle = calle + ", " + ciudad + ", " + estado +
673               " " + cp;
674
675            // obtiene los puntos geográficos para la dirección
676            GeoPoint puntos[] = geoCoder.geoCode( direccionGoogle );
677
678            // si Google Maps no puede encontrar la dirección
679            if ( puntos == null )
680            {
681               error( "El mapa para " + direccionGoogle + " no se pudo encontrar");
682               mapViewer.setRendered( false ); // oculta el mapa
683               return null;
684            } // fin de if
685
686            // centra el mapa para la dirección dada
687            mapViewer_center.setLatitude( puntos[ 0 ].getLatitude() );
688            mapViewer_center.setLongitude( puntos[ 0 ].getLongitude() );
689
690            // crea un marcador para la dirección y establece su texto a mostrar
691            mapMarker.setLatitude( puntos[ 0 ].getLatitude() );
692            mapMarker.setLongitude( puntos[ 0 ].getLongitude() );
693            mapMarker.setMarkup( pnombre + " " + apaterno + "<br/>" + calle +
694               "<br/>" + ciudad + ", " + estado + " " + cp );
695
696            mapViewer.setRendered( true ); // muestra el mapa
697         } // fin de try
698         catch ( Exception e )
699         {
700            error( "Error al procesar busqueda. " + e.getMessage() );
701         } // fin de catch
702
703         return null;
704      } // fin del método buscarBoton_action
705   } // fin de la clase LibretaDirecciones
```

Figura 27.16 | Bean de página que obtiene un mapa para mostrarlo en el componente `MapViewer`. (Parte 4 de 4).

En las líneas 672 y 673 se aplica formato a la dirección como un objeto `String`, para usarlo con la API de Google Maps. En la línea 219 se hace una llamada al método `geoCode` del componente **Geocoding Service Object** con la dirección como argumento. Este método devuelve un arreglo de objetos `GeoPoint` que representen ubicaciones que coincidan con el parámetro dirección. Los objetos `GeoPoint` proporcionan la latitud y longitud de una ubicación dada. Suministramos una dirección completa con la calle, ciudad, estado y código postal como argumento para `geoCode`, por lo que el arreglo devuelto contendrá sólo un objeto `GeoPoint`. En la línea 679 se determina si el arreglo de objetos `GeoPoint` es `null`. De ser así, la dirección no se podría encontrar, y en las líneas 681 a 683 se muestra un mensaje en el componente **Message Group**, informando al usuario sobre el error de búsqueda, se oculta el componente **Map Viewer** y se devuelve `null` para terminar el procesamiento.

En las líneas 687 a 688 se establecen la latitud y la longitud del centro del componente **Map Viewer**, en relación con la latitud y longitud del objeto `GeoPoint` que representa a la dirección seleccionada. En las líneas 691 a 694 se establecen la latitud y longitud del componente **Map Marker**, y se establece el texto a mostrar en el marcador. En la línea 696 se muestra el mapa vuelto a centrar, que contiene el componente **Map Marker** que indica la ubicación del contacto.

En las líneas 698 a 701 se atrapan las excepciones generadas en el cuerpo del método y se muestra un mensaje de error en el componente **Message Group**. Si el usuario sólo seleccionó un nombre de la lista de selecciones en el componente **AutoComplete Text Field**, no habrá errores al buscar en la base de datos, ya que se garantiza que el nombre estará en el formato *apellido paterno*, *primer nombre* apropiado, y estará incluido en la base de datos `LibretaDirecciones`. No incluimos código especial para manejar errores en los casos en que el usuario escribe un nombre que no se puede encontrar en la `LibretaDirecciones`, o para los nombres con formato inapropiado.

27.6 Conclusión

En este capítulo presentamos un ejemplo práctico en tres partes, acerca de cómo crear una aplicación Web que interactúe con una base de datos y proporcione una interacción intensiva con el usuario, mediante el uso de los componentes JSF habilitados para Ajax. Primero le mostramos cómo crear una aplicación `LibretaDirecciones` que permita a un usuario agregar direcciones a la `LibretaDirecciones` y explorar su contenido. Mediante este ejemplo, aprendió a insertar la entrada del usuario en una base de datos Java DB, y a mostrar el contenido de una base de datos en una página Web, mediante el uso de un componente JSF llamado **Tabla**.

Aprendió a descargar e importar la biblioteca de componentes Java BluePrints habilitada para Java. Después extendimos la aplicación `LibretaDirecciones` para incluir un componente **AutoComplete Text Field**. Le mostramos cómo usar una base de datos para mostrar sugerencias en el componente **AutoComplete Text Field**. También aprendió a utilizar formularios virtuales para enviar subconjuntos de los componentes de entrada de un formulario al servidor para procesarlos.

Por último, completamos la tercera parte de la aplicación `LibretaDirecciones` al agregar funcionalidad al formulario de búsqueda. Aprendió a utilizar los componentes **Map Viewer**, **Map Marker** y **Geocoding Service Object** de la biblioteca de componentes Java BluePrints habilitados para Ajax, para mostrar un mapa de Google que indique la ubicación de un contacto.

En el siguiente capítulo, aprenderá a crear y consumir servicios Web con Java. Utilizará el IDE Netbeans 5.5 para crear servicios Web y consumirlos desde aplicaciones de escritorio, y utilizará el IDE Java Studio Creator para consumir un servicio Web desde una aplicación Web. Si prefiere realizar todas estas tareas en un IDE, puede descargar el Netbeans Visual Web Pack 5.5 (`www.netbeans.org/products/visualweb/`) para Netbeans 5.5.

27.7 Recursos Web

`www.deitel.com/ajax/Ajax_resourcecenter.html`

Explore nuestro Centro de recursos Ajax, en donde encontrará vínculos a artículos sobre Ajax, tutoriales, aplicaciones, sitios Web comunitarios y mucho más.

`developers.sun.com/prodtech/javatools/jscreator/learning/tutorials/index.jsp`

Este sitio ofrece docenas de tutoriales acerca de Java Studio Creator 2. De especial interés para este capítulo son las secciones Access Databases (Acceso a bases de datos) y Work with Ajax Components (Trabajo con componentes Ajax).

`developers.sun.com/prodtech/javadb/`

El sitio oficial de Sun sobre Java DB; presenta las generalidades acerca de esta tecnología, y ofrece vínculos a artículos técnicos y un manual acerca del uso de bases de datos Apache Derby.

`java.sun.com/reference/blueprints/`

El sitio de referencia de Sun Developer Network para Java BluePrints.

`blueprints.dev.java.net/`

El sitio de java.net para el proyecto Java BluePrints.

`blueprints.dev.java.net/ajaxcomponents.html`

Información acerca de los componentes habilitados para Ajax que proporciona la biblioteca Java Blueprints.

`developers.sun.com/prodtech/javatools/jscreator/reference/code/samplecomps/index.jsp`

Demuestra los ocho componentes habilitados para Ajax que proporciona la biblioteca Java BluePrints.

`google.com/apis/maps`

Los poseedores de una cuenta de Google pueden registrarse aquí para obtener una clave y utilizar la API de Google Maps.

`ajax.dev.java.net/`

El marco de trabajo del proyecto jMaki Ajax para crear sus propios componentes habilitados para Java.

Resumen

Sección 27.2 Acceso a bases de datos en las aplicaciones Web

- Muchas aplicaciones Web acceden a bases de datos para almacenar y obtener datos persistentes. En esta sección creamos una aplicación Web que utiliza una base de datos Java DB para almacenar contactos en la libreta de direcciones y mostrar contactos de esta libreta en una página Web.
- La página Web permite al usuario introducir nuevos contactos en un formulario. Este formulario consiste en componentes **Campo de texto** para el primer nombre del contacto, su dirección física, ciudad, estado y código postal. El formulario también tiene un botón **Enviar** para enviar los datos al servidor, y un botón **Borrar** para restablecer los campos del formulario. La aplicación almacena la información de la libreta de direcciones en una base de datos llamada `LibretaDirecciones`, la cual tiene una sola tabla llamada `Addresses`. (En el directorio de ejemplos para este capítulo proporcionamos esta base de datos. Puede descargar los ejemplos de `www.deitel.com/books/jhtp7`). Este ejemplo también introduce el componente JSF **Tabla**, el cual muestra las direcciones de la base de datos en formato tabular. En breve le mostraremos cómo configurar el componente **Tabla**.
- El componente **Tabla** da formato y muestra datos de las tablas de una base de datos.
- Cambie la propiedad `title` del componente **Tabla** para especificar el texto que se va a mostrar en la parte superior de la **Tabla**.
- Para utilizar una base de datos en una aplicación Web de Java Studio Creator 2, primero debemos iniciar el servidor de bases de datos integrado del IDE, el cual incluye controladores para muchos tipos de bases de datos, incluyendo Java DB.
- Para iniciar el servidor, haga clic en la ficha **Servidores** debajo del menú **Archivo**, haga clic con el botón derecho en **Servidor Bundled Database** en la parte inferior de la ventana **Servidores** y seleccione **Iniciar Bundled Database**.
- Para agregar una base de datos Java DB a un proyecto, haga clic con el botón derecho en el nodo **Orígenes de datos** en la parte superior de la ventana **Servidores** y seleccione **Agregar origen de datos…**. En el cuadro de diálogo **Agregar origen de datos**, escriba el nombre del origen de datos y seleccione **Derby** en el tipo de servidor. Especifique el ID de usuario y la contraseña para la base de datos. Para el URL de la base de datos, escriba `jdbc:derby://localhost:21527/`*NombreDeSuBaseDeDatos*. Este URL indica que la base de datos reside en el equipo local y acepta conexiones en el puerto 21527. Haga clic en el botón **Seleccionar** para elegir una tabla que se utilizará para validar la base de datos. Haga clic en **Seleccionar** para cerrar este cuadro de diálogo, y después haga clic en **Agregar** para agregar la base de datos como origen de datos para el proyecto y cierre el cuadro de diálogo.
- Para configurar un componente **Tabla** para mostrar los datos de una base de datos, haga clic con el botón derecho del ratón en el componente **Tabla** y seleccione **Enlazar con datos** para mostrar el cuadro de diálogo **Diseño de tabla**. Haga clic en el botón **Agregar proveedor de datos…** para mostrar el cuadro de diálogo **Agregar proveedor de datos**, el cual contiene una lista de los orígenes de datos disponibles. Expanda el nodo de la base de datos, expanda el nodo **Tablas**, seleccione una tabla y haga clic en **Agregar**. El cuadro de diálogo **Diseño de tabla** mostrará una lista de las columnas en la tabla de la base de datos. Todos los elementos bajo el encabezado **Seleccionado** se mostrarán en la **Tabla**. Para eliminar una columna de la **Tabla**, puede seleccionarla y hacer clic en el botón **<**.
- De manera predeterminada, la **Tabla** utiliza los nombres de las columnas de la tabla de la base de datos en mayúsculas como encabezados. Para modificar estos encabezados, seleccione una columna y edite su propiedad `headerText` en la ventana **Propiedades**. Para seleccionar una columna, expanda el nodo de la tabla en la ventana **Esquema** (estando en modo **Diseño**) y después seleccione el objeto columna apropiado.
- Al hacer clic en la casilla de verificación a un lado de la propiedad `paginationControl` de la tabla en la ventana **Propiedades**, se configura esta **Tabla** para paginación automática. Se mostrarán cinco filas a la vez, y se agregarán botones para avanzar hacia delante y hacia atrás, entre grupos de cinco contactos, al final de la **Tabla**.
- Al enlazar un componente **Tabla** con un proveedor de datos, se agrega un nuevo objeto `CachedRowSetDataProvider` al nodo de la aplicación en la ventana **Esquema**. Un objeto `CachedRowSetDataProvider` proporciona un objeto `RowSet` desplazable que puede enlazarse con un componente **Tabla** para mostrar los datos del objeto `RowSet`.
- El objeto `CachedRowSet` envuelto por nuestro objeto `addressesDataProvider` está configurado de manera predeterminada para ejecutar una consulta SQL que seleccione todos los datos en la tabla de la base de datos. Para editar esta consulta SQL, puede expandir el nodo `SessionBean` en la ventana **Esquema** y hacer doble clic en el elemento `RowSet` para abrir la ventana del editor de consultas.
- Cada fila en un objeto `CachedRowSetDataProvider` tiene su propia clave; el método `appendRow`, que agrega una nueva fila al objeto `CachedRowSet`, devuelve la clave para la nueva fila.
- El método `commitChanges` de la clase `CachedRowSetDataProvider` aplica los cambios en el objeto `CachedRowSet` a la base de datos.

- El método `refresh` de `CachedRowSetDataProvider` vuelve a ejecutar la instrucción SQL del objeto `CachedRowSet` envuelto.

Sección 27.3 Componentes JSF habilitados para Ajax

- El término Ajax (JavaScript y XML asíncronos) fue ideado por Jesse James Garrett de Adaptive Path, Inc. en febrero de 2005, para describir un rango de tecnologías para desarrollar aplicaciones Web dinámicas y con gran capacidad de respuesta.
- Ajax separa la parte correspondiente a la interacción con el usuario de una aplicación, de la interacción con su servidor, permitiendo que ambas procedan en forma asincrónica y en paralelo. Esto permite a las aplicaciones Ajax basadas en Web ejecutarse a velocidades que se asemejan a las de las aplicaciones de escritorio.
- Ajax realiza llamadas asíncronas al servidor para intercambiar pequeñas cantidades de datos con cada llamada.
- Ajax permite que se vuelvan a cargar sólo las porciones necesarias de la página, lo cual ahorra tiempo y recursos.
- Las aplicaciones Ajax contienen marcado de XHTML y CSS al igual que cualquier otra página Web, y hacen uso de las tecnologías de secuencias de comandos del lado cliente (como JavaScript) para interactuar con los elementos de las páginas.
- El objeto `XMLHttpRequestObject` permite los intercambios asíncronos con el servidor Web, lo cual hace que las aplicaciones Ajax tengan una gran capacidad de respuesta. Este objeto se puede utilizar en la mayoría de los lenguajes de secuencias de comandos para pasar datos XML del cliente al servidor, y para procesar los datos XML que el servidor envía de vuelta al cliente.
- Las bibliotecas Ajax facilitan el proceso de aprovechar los beneficios de Ajax en las aplicaciones Web, sin tener que escribir Ajax "puro".
- La biblioteca de componentes Java BluePrints habilitados para Ajax proporciona componentes JSF habilitados para Ajax.
- Para utilizar los componentes Java BluePrints habilitados para Ajax en Java Studio Creator 2, debe descargarlos e importarlos. El IDE proporciona un asistente para instalar este grupo de componentes. Seleccione **Herramientas > Centro de actualización** para mostrar el cuadro de diálogo **Asistente del centro de actualización**. Haga clic en **Siguiente >** para buscar actualizaciones disponibles. En el área **Nuevos módulos y actualizaciones disponibles** del cuadro de diálogo, seleccione **BluePrints AJAX Components** y haga clic con el botón derecho del ratón en el botón de flecha (**>**) para agregarlo a la lista de elementos que desea instalar. Haga clic en **Siguiente >** y siga los indicadores para aceptar las condiciones de uso y descargar los componentes. Cuando se complete la descarga, haga clic en **Siguiente** y luego en **Terminar**. Haga clic en **Aceptar** para reiniciar el IDE.
- A continuación, debe importar los componentes en la **Paleta**. Seleccione **Herramientas > Administrador de bibliotecas de componentes** y después haga clic en **Importar…**. Haga clic en el botón **Examinar…** en el cuadro de diálogo **Importar biblioteca de componentes** que aparezca. Seleccione el archivo `ui.complib` y haga clic en **Abrir**. Haga clic en **Aceptar** para importar los componentes **BluePrints AJAX Components** y **BluePrints AJAX SupportBeans**.

*Sección 27.4 **AutoComplete Text Field** y formularios virtuales*

- El componente **AutoComplete Text Field** proporciona una lista de sugerencias de un origen de datos (como una base de datos o un servicio Web) a medida que el usuario escribe.
- Los formularios virtuales se utilizan cuando deseamos que un botón envíe un subconjunto de los campos de entrada de la página al servidor.
- Los formularios virtuales se utilizan cuando deseamos que un botón envíe un subconjunto de los campos de entrada de la página al servidor.
- Los formularios virtuales nos permiten mostrar varios formularios en la misma página. Nos permiten especificar un emisor y uno o más participantes para cada formulario. Al hacer clic en el componente emisor del formulario virtual, sólo se envían al servidor los valores de sus componentes participantes.
- Para agregar formularios virtuales a la página, haga clic con el botón derecho en el componente emisor que se encuentra en el formulario, y seleccione **Configurar formularios virtuales…** en el menú contextual para que aparezca el cuadro de diálogo **Configurar formularios virtuales**. Haga clic en **Nuevo** para agregar un formulario virtual; después haga clic en la columna **Nombre** y especifique el nombre del nuevo formulario. Haga doble clic en la columna **Enviar** y cambie la opción a **Sí** para indicar que este botón se debe utilizar para enviar el formulario virtual **agregarForm**. Haga clic en **Aceptar** para salir del cuadro de diálogo. Después, seleccione todos los componentes de entrada que participarán en el formulario virtual. Haga clic con el botón derecho del ratón en uno de los componentes seleccionados y elija la opción **Configurar formularios virtuales…**. En la columna **Función** del formulario virtual apropiado, cambie la opción a **Sí** para indicar que los valores en estos componentes deben enviarse al servidor cuando se envíe el formulario.

- Para ver los formularios virtuales en modo **Diseño**, haga clic en el botón **Mostrar formularios virtuales** ( ) en la parte superior del panel Diseñador visual para mostrar una leyenda de los formularios virtuales en la página.
- El atributo `completionMethod` de un componente **AutoComplete Text Field** está enlazado al manejador de eventos `complete` de un bean de página. Para crear este método, haga clic con el botón derecho en el componente **AutoComplete Text Field** en vista **Diseño** y seleccione **Editar controlador de eventos > complete**.
- El manejador de eventos `complete` se invoca después de cada pulsación de tecla en un componente **AutoComplete Text Field** para actualizar la lista de sugerencias con base en el texto que el usuario ha escrito hasta cierto punto. El método recibe una cadena que contiene el texto introducido por el usuario, junto con un objeto `CompletionResult` que se utiliza para mostrar sugerencias al usuario.

Sección 27.5 Componente **Map Viewer** *de Google Maps*

- Un componente **Map Viewer** habilitado para Ajax utiliza el servicio Web de la API de Google Maps para buscar y mostrar mapas. Un componente **Map Marker** apunta a una ubicación en un mapa.
- Para utilizar el componente **Map Viewer**, debe tener una cuenta con Google. Visite el sitio `https://www.google.com/accounts/ManageAccount` para registrarse y obtener una cuenta gratuita. Debe obtener una clave para usar la API de Google Maps en `www.google.com/apis/maps`. La clave que reciba será específica para su aplicación Web y limitará el número de mapas que puede mostrar la aplicación por día. Cuando se registre para la clave, tendrá que escribir el URL para la aplicación que utilizará la API de Google Maps. Si va a desplegar la aplicación sólo en el servidor de prueba integrado de Java Studio Creator 2, escriba el URL `http://localhost:29080`.
- Para utilizar un componente **Map Viewer**, establezca su propiedad **key** con la clave de la API de Google Maps que obtuvo.
- Un componente **Map Marker** (de la sección **AJAX Support Beans** de la **Paleta**) marca una ubicación en un mapa. Debe enlazar el marcador con el mapa, de manera que el marcador se pueda mostrar en el mapa. Para ello, haga clic con el botón derecho en el componente **Map Viewer** en modo **Diseño** y seleccione **Enlaces de propiedades…** para mostrar el cuadro de diálogo **Enlaces de propiedades**. Seleccione **info** de la columna **Seleccionar propiedad enlazable** del cuadro de diálogo, y después seleccione **mapMarker** de la columna **Seleccionar destino de enlace**. Haga clic en **Aplicar** y después en **Cerrar**.
- Un componente **Geocoding Service Object** (de la sección **AJAX Support Beans** de la **Paleta**) convierte las direcciones de las calles en latitudes y longitudes que el componente **Map Viewer** utiliza para mostrar un mapa apropiado.
- El atributo **center** del componente **Map Viewer** está enlazado a la propiedad `mapViewer_center` del bean de página. Esta propiedad se manipula en el archivo de bean de página, para centrar el mapa en la dirección deseada.
- El método `geoCode` del componente **Geocoding Service Object** recibe una dirección como un argumento y devuelve un arreglo de objetos `GeoPoint` que representan ubicaciones, las cuales coinciden con el parámetro dirección. Los objetos `GeoPoint` proporcionan la latitud y la longitud de una ubicación dada.

Terminología

Ajax (JavaScript y XML asíncronos)
Apache Derby
API de Google Maps
AutoComplete Text Field, componente JSF
biblioteca de componentes Java BluePrints habilitados para Ajax
bibliotecas de componentes habilitadas para Ajax
Button, componente JSF
Buy Now Button, componente JSF
`CachedRowSet`, interfaz
`CachedRowSetDataProvider`, clase
ciclo de vida del procesamiento de eventos
`commitChanges`, método de la clase `CachedRowSetDataProvider`
componentes JSF habilitados para Ajax
elemento JSF
emisor en un formulario virtual
enlazar una **Tabla** JSF con la tabla de una base de datos
formulario virtual
`geoCode`, método de un componente **Geocoding Service Object**
Geocoding Service Object, componente
Google Maps
Java BluePrints
Java DB
JavaServer Faces (JSF)
Jesse James Garrett
Map Marker, componente JSF
Map Viewer, componente JSF
Message Group, componente JSF
participante en un formulario virtual
Popup Calendar, componente JSF
`primary`, propiedad de un **Botón** JSF
Progress Bar, componente JSF
proveedor de datos
Rating, componente JSF
`refresh`, método de `class CachedRowSetDataProvider`

`reset`, propiedad de un **Botón** JSF
Rich Textarea Editor, componente JSF
Select Value Text Field, componente JSF
servidor de bases de datos integrado
Servidores, ficha en Java Studio Creator 2
Tabla, componente JSF
`ui:staticText`, elemento JSF
`ui:table`, elemento JSF
`ui:tableRowGroup`, elemento JSF
XMLHttpRequestObject

Ejercicios de autoevaluación

27.1 Conteste con *verdadero* o *falso* a cada una de las siguientes proposiciones; en caso de ser *falso*, explique por qué.

a) El componente JSF **Tabla** nos permite desplegar otros componentes y texto en formato tabular.
b) Los formularios virtuales permiten mostrar varios formularios (cada uno con su botón **Enviar**) en la misma página Web.
c) Un componente `CachedRowSetDataProvider` se almacena en el objeto `SessionBean` y ejecuta consultas SQL para proporcionar componentes **Tabla** con datos a mostrar.
d) El objeto `XMLHttpRequestObject` proporciona acceso al objeto petición de una página.
e) El manejador de eventos `complete` para un componente **AutoComplete Text Field** se llama después de cada pulsación de tecla en el campo de texto, para proporcionar una lista de sugerencias con base en lo que ya se ha escrito.
f) Un proveedor de datos vuelve a ejecutar automáticamente su comando SQL para proporcionar información actualizada de la base de datos en cada actualización de página.
g) Para volver a centrar un componente **Map Viewer**, debe establecer la longitud y latitud del centro del mapa.

27.2 Complete los siguientes enunciados.

a) Ajax es un acrónimo para ______________.
b) El método ______________ de la clase ______________ actualiza una base de datos para reflejar los cambios realizados en el proveedor de datos de la base de datos.
c) Un ______________ es un componente de soporte utilizado para traducir direcciones en latitudes y longitudes, para mostrarlas en un componente **Map Viewer**.
d) Un formulario virtual especifica que ciertos componentes JSF son ______________ cuyos datos se enviarán cuando se haga clic en el componente emisor.
e) Los componentes Ajax para Java Studio Creator 2, como **AutoComplete Text Field** y **Map Viewer**, son proporcionados por ______________.

Respuestas a los ejercicios de autoevaluación

27.1 a) Falso. Los componentes `Tabla` se utilizan para mostrar datos de las bases de datos. b) Verdadero. c) Falso. El componente `CachedRowSetDataProvider` es una propiedad del bean de página. Envuelve un objeto `CachedRowSet`, el cual se almacena en el objeto `SessionBean` y ejecuta consultas SQL. d) Falso. El objeto `XMLHttpRequestObject` es un objeto que permite intercambios asincrónicos con un servidor Web. e) Verdadero. f) Falso. Debe llamar al método `refresh` en el proveedor de datos para volver a ejecutar el comando SQL. g) Verdadero.

27.2 a) JavaScript y XML asíncronos. b) `commitChanges`, `CachedRowSetDataProvider`. c) **Geocoding Service Object**. d) participantes. e) La biblioteca de componentes Java BluePrint habilitados para Ajax.

Ejercicios

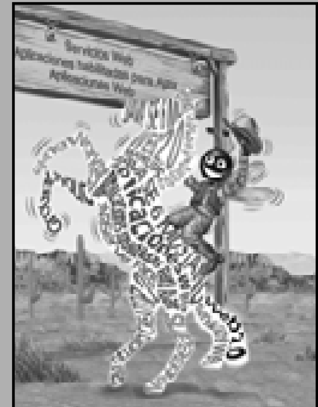
27.3 *(Aplicación LibroVisitantes)* Cree una página Web JSF que permita a los usuarios registrarse en un libro de visitantes y verlo. Use la base de datos `LibroVisitantes` (que se proporciona en el directorio de ejemplos para este capítulo) para almacenar las entradas en el libro de visitantes. La base de datos `LibroVisitantes` tiene una sola tabla llamada `Messages`, la cual contiene cuatro columnas: `date`, `name`, `email` y `message`. La base de datos contiene unas cuantas entradas de ejemplo. En la página Web, proporcione componentes **Campo de texto** para el nombre del usuario y la dirección de correo electrónico, y un componente **Área de texto** para el mensaje. Agregue un **Botón Enviar** y un componente **Tabla**, y configure la **Tabla** para mostrar las entradas en el libro de visitantes. Use el método manejador de acciones del **Botón Enviar** para insertar una nueva fila que contenga la entrada del usuario y la fecha de hoy en la base de datos `LibroVisitantes`.

27.4 *(Modificación a la aplicación LibretaDirecciones)* Modifique la aplicación `LibretaDirecciones`, de manera que los usuarios introduzcan búsquedas en el componente **AutoComplete TextField**, en el formato *primer nombre*, *apellido paterno*. Necesitará agregar un nuevo proveedor de datos (o modificar el existente) para ordenar las filas en la base de datos `LibretaDirecciones` por primer nombre, y después por apellido paterno.

27.5 *(Aplicación de búsqueda en mapas)* Cree una página Web JSF que permita a los usuarios obtener un mapa de cualquier dirección. Recuerde que la búsqueda de una ubicación mediante la API de Google Maps devuelve un arreglo de objetos `GeoPoint`. Busque las ubicaciones que introduzca el usuario en un **Campo de texto**, y muestre un mapa en la primera ubicación del arreglo `GeoPoint` resultante. Para manejar varios resultados de búsqueda, muestre todos los resultados en un componente **Cuadro de lista**. Para obtener una representación de cadena de cada resultado, invoque el método `toString` en un objeto `GeoPoint`. Agregue un **Botón** que permita a los usuarios seleccionar un resultado del **Cuadro de lista** y muestre un mapa para ese resultado con un componente **Map Marker** que muestre la ubicación en el mapa. Por último, utilice un componente **Grupo de mensaje**s para mostrar los mensajes relacionados con los errores de búsqueda. En caso de un error, y cuando la página se cargue por primera vez, vuelva a centrar el mapa en una ubicación predeterminada de su preferencia.

28

Servicios Web JAX-WS, Web 2.0 y Mash-ups



Un cliente para mí es una simple unidad, un factor en un problema.

—Sir Arthur Conan Doyle

También sirven a aquéllos que sólo permanecen de pie y esperan.

—John Milton

…si las cosas más simples de la naturaleza tienen un mensaje que usted comprende, regocíjese, porque su alma está viva.

—Eleonora Duse

El protocolo es todo.

—Francoise Giuliani

OBJETIVOS

En este capítulo aprenderá a:

- Conocer qué es un servicio Web.
- Publicar y consumir los servicios Web en Netbeans.
- Conocer los elementos que conforman los servicios Web, como las descripciones de servicios y las clases que implementan estos servicios.
- Crear aplicaciones de escritorio cliente y Web que invoquen métodos de un servicio Web.
- Conocer la parte importante que desempeñan XML y el Protocolo simple de acceso a objetos (SOAP) para habilitar los servicios Web.
- Usar el rastreo de sesiones en los servicios Web para mantener la información de estado de los clientes.
- Usar JDBC con los servicios Web para conectarse a las bases de datos.
- Pasar objetos de tipos definidos por el usuario hacia y desde un servicio Web.

Plan general

28.1 Introducción

En este capítulo presentaremos los servicios Web, los cuales promueven la portabilidad y reutilización de software en aplicaciones que operan a través de Internet. Un **servicio Web** es un componente de software almacenado en una computadora, el cual se puede utilizar mediante llamadas a métodos desde una aplicación (u otro componente de software) en otra computadora, a través de una red. Los servicios Web se comunican mediante el uso de tecnologías como XML y HTTP. Varias APIs de Java facilitan los servicios Web. En este capítulo trataremos con APIs basadas en el **Protocolo simple de acceso a objetos (SOAP)**: un protocolo basado en XML que permite la comunicación entre los clientes y los servicios Web, aun cuando el cliente y el servicio Web estén escritos en distintos lenguajes. Hay otras tecnologías de servicios Web, como la Transferencia representativa de estado (REST), que no veremos en este capítulo. Para obtener información acerca de REST, consulte los recursos Web de la sección 28.10 y visite nuestro Centro de recursos sobre servicios Web en

```
www.deitel.com/WebServices
```

La Biblioteca Deitel de contenido gratuito (Deitel Free Content Library) incluye los siguientes tutoriales de introducción a XML:

```
www.deitel.com/articles/xml_tutorials/20060401/XMLBasics/
www.deitel.com/articles/xml_tutorials/20060401/XMLStructuringData/
```

Los servicios Web tienen grandes implicaciones para las **transacciones de negocio a negocio (B2B)**. Permiten a los negocios realizar transacciones a través de servicios Web estandarizados, con una amplia disponibilidad,

en vez de depender de aplicaciones propietarias. Los servicios Web y SOAP son independientes de la plataforma y del lenguaje, por lo que las compañías pueden colaborar a través de servicios Web sin tener que preocuparse por la compatibilidad de sus tecnologías de hardware, software y comunicaciones. Compañías como Amazon, Google, eBay, PayPal y muchas otras están usando servicios Web para su beneficio, al facilitar sus aplicaciones del lado servidor a sus socios, a través de los servicios Web.

Al comprar servicios Web y utilizar los diversos servicios Web gratuitos que son importantes para sus actividades comerciales, las compañías invierten menor tiempo en desarrollar nuevas aplicaciones y pueden crear nuevas aplicaciones innovadoras. Los comercios electrónicos pueden usar servicios Web para proporcionar a sus clientes experiencias de compra mejoradas. Consideremos el caso de una tienda de música en línea. El sitio Web de la tienda ofrece vínculos a información sobre varios CDs, lo cual permite a los usuarios comprar los CDs, saber acerca de los artistas, buscar más títulos de ellos, buscar música de otros artistas que los usuarios puedan disfrutar, y mucho más. Otra compañía que vende boletos para conciertos proporciona un servicio Web que muestra las fechas de los próximos conciertos de varios artistas, y permite a los usuarios comprar boletos. Al consumir el servicio Web para boletos de conciertos en su sitio, la tienda de música en línea puede proporcionar un servicio adicional a sus clientes e incrementar el tráfico de su sitio, y tal vez obtener una comisión por las ventas de los boletos de conciertos. La compañía que vende boletos de conciertos también se beneficia de la relación comercial al vender más boletos y quizá al recibir ingresos de la tienda de música en línea por el uso de su servicio Web.

Cualquier programador de Java con un conocimiento sobre los servicios Web puede escribir aplicaciones que puedan "consumir" servicios Web. Las aplicaciones resultantes llamarían a los métodos de los servicios Web de objetos que se ejecuten en servidores, que podrían estar a miles de kilómetros de distancia. Para saber más acerca de los servicios Web, lea las Generalidades sobre la tecnología Java y los servicios Web (Java Technology and Web Services Overview) en `java.sun.com/webservices/overview.html`.

Netbeans 5.5 y Sun Java Studio Creator 2

Netbeans 5.5 y Sun Java Studio Creator 2 (ambos desarrollados por Sun) son dos de las muchas herramientas que permiten a los programadores "publicar" y "consumir" servicios Web. Vamos a demostrar cómo usar estas herramientas para implementar servicios Web e invocarlos desde aplicaciones cliente. Para cada ejemplo, proporcionaremos el código para el servicio Web, y después presentaremos una aplicación cliente que utiliza el servicio Web. En nuestros primeros ejemplos vamos a crear servicios Web y aplicaciones cliente en Netbeans. Después demostraremos servicios Web que utilizan características más sofisticadas, como la manipulación de bases de datos con JDBC (que presentamos en el capítulo 25, Acceso a bases de datos con JDBC) y la manipulación de objetos de clases.

Sun Java Studio Creator 2 facilita el desarrollo de aplicaciones Web y el consumo de servicios Web. Netbeans proporciona aún más herramientas, incluyendo la habilidad de crear, publicar y consumir servicios Web. Utilizamos Netbeans para crear y publicar servicios Web, y para crear aplicaciones de escritorio que los consuman. Utilizamos Sun Java Studio Creator 2 para crear aplicaciones Web que consuman servicios Web. Las herramientas de Sun Java Studio Creator 2 se pueden agregar a Netbeans mediante el Netbeans Visual Web Pack. Para obtener más información acerca de este complemento de Netbeans, visite `www.netbeans.org/products/visualweb/`.

28.1.1 Descarga, instalación y configuración de Netbeans 5.5 y Sun Java System Application Server

Para desarrollar los servicios Web en este capítulo, utilizamos Netbeans 5.5 y Sun Java System Application Server con las opciones de instalación predeterminadas. El sitio Web de Netbeans proporciona un instalador integrado para ambos productos. Para descargar el instalador, visite el sitio

```
www.netbeans.org/products/ide/
```

y después haga clic en el botón **Download Netbeans IDE**. En la siguiente página, seleccione su sistema operativo y lenguaje, y después siga las instrucciones que aparecen en pantalla. Es posible que para cuando usted entre, ya haya una versión más reciente.

Una vez que haya descargado e instalado estas herramientas, ejecute el IDE Netbeans. En Windows XP, el instalador colocará una entrada para Netbeans en **Inicio > Todos los programas**. Antes de proceder con el resto del capítulo, realice los siguientes pasos para configurar Netbeans y permitir realizar pruebas con Sun Java System Application Server:

1. Seleccione **Tools > Server Manager** para mostrar el cuadro de diálogo **Server Manager**. Si ya aparece Sun Java System Application Server en la lista de servidores, omita los *pasos del 2 al 6*.
2. Haga clic en el botón **Add Server...** en la esquina superior izquierda del cuadro de diálogo para que aparezca el cuadro de diálogo **Add Server Instance**.
3. Seleccione **Sun Java System Application Server** de la lista desplegable **Server**, y después haga clic en **Next >**.
4. En el campo **Platform Location**, especifique la ubicación de instalación de Sun Java System Application Server; el valor predeterminado es `C:\Sun\AppServer` en Windows. Haga clic en **Next >**.
5. Especifique el nombre de usuario y la contraseña para el servidor; los valores predeterminados son `admin` y `adminadmin`, respectivamente. Haga clic en **Finish**.
6. Haga clic en **Close** para cerrar el cuadro de diálogo **Server Manager**.

28.1.2 Centro de recursos de servicios Web y Centros de recursos sobre Java en `www.deitel.com`

Visite nuestro Centro de recursos de servicios Web en `www.deitel.com/WebServices/` para obtener información acerca de cómo diseñar e implementar servicios Web en muchos lenguajes, y para obtener información acerca de los servicios Web que ofrecen compañías como Google, Amazon y eBay. También encontrará muchas herramientas adicionales de Java para publicar y consumir servicios Web.

Nuestros Centros de recursos sobre Java en

```
www.deitel.com/java/
http://www.deitel.com/ResourceCenters/Programming/JavaSE6/tabid/1056/Default.aspx
www.deitel.com/JavaEE5/
```

ofrecen información adicional específica sobre Java, como libros, informes, artículos, diarios, sitios Web y blogs que abarcan una gran variedad de temas sobre Java (incluyendo los servicios Web de Java).

28.2 Fundamentos de los servicios Web de Java

La computadora en la que reside un servicio Web se conoce como **equipo remoto** o **servidor**. La aplicación (es decir, el cliente) que accede al servicio Web envía la llamada a un método a través de una red al equipo remoto, el cual procesa la llamada y devuelve una respuesta a la aplicación, a través de la red. Este tipo de computación distribuida es benéfica en muchas aplicaciones. Por ejemplo, una aplicación cliente sin acceso directo a una base de datos en un servidor remoto podría obtener los datos a través de un servicio Web. De manera similar, una aplicación que carezca del poder para realizar ciertos cálculos podría usar un servicio Web para aprovechar los recursos superiores de otro sistema.

En Java, un servicio Web se implementa como una clase. En capítulos anteriores, todas las piezas de una aplicación residían en un equipo. La clase que representa el servicio Web reside en un servidor; no forma parte de la aplicación cliente.

Al proceso de hacer que un servicio Web esté disponible para recibir peticiones de los clientes se le conoce como **publicación de un servicio Web**; al proceso de utilizar un servicio Web desde una aplicación cliente se le conoce como **consumo de un servicio Web**. Una aplicación que consume un servicio Web consiste de dos partes: un objeto de una **clase proxy** para interactuar con el servicio Web y una aplicación cliente que consume el servicio Web, invocando a los métodos en el objeto de la clase proxy. El código cliente invoca los métodos en el objeto proxy, el cual se encarga de los detalles de la comunicación con el servicio Web (como el paso de los argumentos a los métodos del servicio Web y la recepción de los valores de retorno del servicio Web) por el cliente. Esta comunicación se puede llevar a cabo a través de una red local, de Internet o inclusive con un servicio Web en la misma computadora. El servicio Web realiza la tarea correspondiente y devuelve los resultados al objeto proxy, el cual devuelve entonces los resultados al código cliente. En la figura 28.1 se muestran las interacciones entre el código cliente, la clase proxy y el servicio Web. Como pronto veremos, Netbeans y Sun Java Studio Creator 2 crean estas clases proxy por usted en sus aplicaciones cliente.

Las peticiones a los servicios Web, y las respuestas de éste que se crean con **JAX-WS 2.0** (el marco de trabajo más reciente de servicios Web de Java) se transmiten, por lo regular, a través de SOAP. Cualquier cliente capaz de generar y procesar mensajes SOAP puede interactuar con un servicio Web, sin importar el lenguaje en el que esté escrito. En la sección 28.5 hablaremos sobre SOAP.

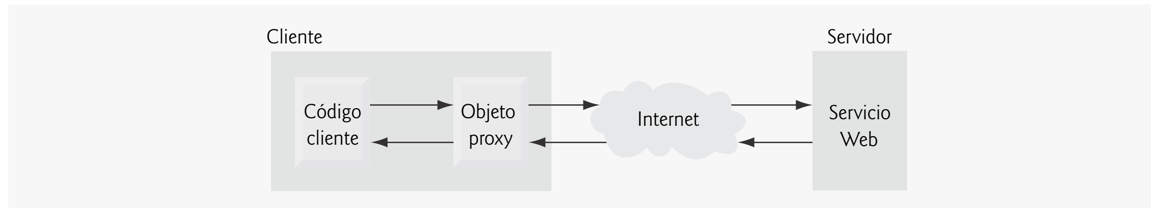


Figura 28.1 | Interacción entre un servicio Web y su cliente.

28.3 Creación, publicación, prueba y descripción de un servicio Web

Las siguientes subsecciones demuestran cómo crear, publicar y probar un servicio Web llamado `EnteroEnorme`, el cual realiza cálculos con enteros positivos de hasta 100 dígitos de longitud (que se mantienen como arreglos de dígitos). Dichos enteros son mucho más grandes de lo que los tipos primitivos integrales de Java pueden representar. El servicio Web `EnteroEnorme` proporciona métodos que reciben dos "enteros enormes" (representados como objetos `String`) y determinan su suma, su diferencia, cuál es mayor, cuál es menor o si los dos números son iguales. Estos métodos serán servicios disponibles para otras aplicaciones a través de la Web (de ahí que se les llame servicios Web).

28.3.1 Creación de un proyecto de aplicación Web y cómo agregar una clase de servicio Web en Netbeans

Al crear un servicio Web en Netbeans, nos enfocamos en la lógica del servicio Web y dejamos que el IDE se encargue de la infraestructura del servicio Web. Para crear un servicio Web en Netbeans, primero debemos crear un proyecto de tipo **`aplicación Web (Web Application)`**. Netbeans utiliza este tipo de proyecto para las aplicaciones Web que se ejecutan en clientes basados en navegador, y para los servicios Web invocados por otras aplicaciones.

Creación de un proyecto de `aplicación Web` en Netbeans 5.5

Para crear una aplicación Web, siga los siguientes pasos:

1. Seleccione **File > New Project** para abrir el cuadro de diálogo **New Project**.
2. Seleccione **Web** de la lista **Categories** del cuadro de diálogo, y después seleccione **Web Application** de la lista **Projects**. Haga clic en **Next >**.
3. Especifique el nombre de su proyecto (`EnteroEnorme`) en el campo **Project Name**, y en el campo **Project Location** especifique en dónde le gustaría guardar el proyecto. Puede hacer clic en el botón **Browse** para seleccionar la ubicación.
4. Seleccione **Sun Java System Application Server** de la lista desplegable **Server**.
5. Seleccione **Java EE 5** de la lista desplegable **J2EE Version**.
6. Haga clic en **Finish** para cerrar el cuadro de diálogo **New Project**.

Esto creará una aplicación Web que se ejecutará en un navegador Web. Al crear una **aplicación Web (Web Application)** en Netbeans, el IDE genera archivos adicionales que ofrecen soporte a la aplicación Web. En este capítulo, hablaremos sólo sobre los archivos específicos para los servicios Web.

Cómo agregar una clase de servicio Web a un proyecto de `Aplicación Web`

Para crear un servicio Web, realice los siguientes pasos para agregar una clase de servicio Web al proyecto:

1. En la ficha **Projects** en Netbeans (justo debajo del menú **File**), haga clic con el botón derecho del ratón en el nodo del proyecto **EnteroEnorme** y seleccione **New > Web Service...** para que aparezca el cuadro de diálogo **New Web Service**.
2. Especifique `EnteroEnorme` en el campo **Web Service Name**.
3. Especifique `com.deitel.jhtp7.cap28.enteroenorme` en el campo **Package**.
4. Haga clic en **Finish** para cerrar el cuadro de diálogo **New Web Service**.

El IDE genera una clase de servicio Web de ejemplo con el nombre que especificó en el *paso 2*. En esta clase, definirá los métodos que su servicio Web tiene disponibles para las aplicaciones cliente. Cuando en cierto momento genere su aplicación, el IDE generará otros archivos de soporte (que veremos en breve) para su servicio Web.

28.3.2 Definición del servicio Web `EnteroEnorme` en Netbeans

La figura 28.2 contiene el código del servicio Web `EnteroEnorme`. Puede implementar este código por su cuenta en el archivo `EnteroEnorme.java` que se creó en la sección 28.3.1, o puede simplemente sustituir el código en `EnteroEnorme.java` con una copia de nuestro código de la carpeta de este ejemplo. Encontrará este archivo en la carpeta del proyecto `src\java\com\deitel\jhtp7\cap28\enteroenorme`. Puede descargar los ejemplos del libro en `www.deitel.com/books/jhtp7`.

```java
// Fig. 28.2: EnteroEnorme.java
// Servicio Web EnteroEnorme que realiza operaciones con enteros grandes.
package com.deitel.jhtp7.ch28.enteroenorme;

import javax.jws.WebService; // el programa usa la anotación @WebService
import javax.jws.WebMethod; // el programa usa la anotación @WebMethod
import javax.jws.WebParam; // el programa usa la anotación @WebParam

@WebService( // anota la clase como un servicio Web
   name = "EnteroEnorme", // establece el nombre de la clase
   serviceName = "ServicioEnteroEnorme" ) // establece el nombre del servicio
public class EnteroEnorme
{
   private final static int MAXIMO = 100; // máximo número de dígitos
   public int[] numero = new int[ MAXIMO ]; // almacena el entero enorme

   // devuelve una representación String de un EnteroEnorme
   public String toString()
   {
      String valor = "";

      // convierte EnteroEnorme en un objeto String
      for ( int digito : numero )
         valor = digito + valor; // coloca el siguente dígito al principio del valor

      // localiza la posición del primer dígito distinto de cero
      int longitud = valor.length();
      int posicion = -1;

      for ( int i = 0; i < longitud; i++ )
      {
         if ( valor.charAt( i ) != '0' )
```

Figura 28.2 | Servicio Web `EnteroEnorme` que realiza operaciones con enteros grandes. (Parte 1 de 3).

```
         {
            posicion = i; // primer dígito distinto de cero
            break;
         }
      } // fin de for

      return ( posicion != -1 ? valor.substring( posicion ) : "0" );
   } // fin del método toString

   // crea un objeto EnteroEnorme a partir de un objeto String
   public static EnteroEnorme analizarEnteroEnorme( String s )
   {
      EnteroEnorme temp = new EnteroEnorme();
      int tamanio = s.length();

      for ( int i = 0; i < tamanio; i++ )
         temp.numero[ i ] = s.charAt( tamanio - i - 1 ) - '0';

      return temp;
   } // fin del método analizarEnteroEnorme

   // Método Web que suma enteros enormes representados por argumentos String
   @WebMethod( operationName = "sumar" )
   public String sumar( @WebParam( name = "primero" ) String primero,
      @WebParam( name = "segundo" ) String segundo )
   {
      int acarreo = 0; // el valor que se va a acarrear
      EnteroEnorme operando1 = EnteroEnorme.analizarEnteroEnorme( primero );
      EnteroEnorme operando2 = EnteroEnorme.analizarEnteroEnorme( segundo );
      EnteroEnorme resultado = new EnteroEnorme(); // almacena el resultado de la suma

      // realiza la suma en cada dígito
      for ( int i = 0; i < MAXIMO; i++ )
      {
         // suma los dígitos correspondientes en cada número y el valor acarreado;
         // almacena el resultado en la columna correspondiente del resultado EnteroEnorme
         resultado.numero[ i ] =
            ( operando1.numero[ i ] + operando2.numero[ i ] + acarreo ) % 10;

         // establece el acarreo para la siguiente columna
         acarreo =
            ( operando1.numero[ i ] + operando2.numero[ i ] + acarreo ) / 10;
      } // fin de for

      return resultado.toString();
   } // fin del Método Web sumar

   // Método Web que resta los enteros representados por argumentos String
   @WebMethod( operationName = "restar" )
   public String restar( @WebParam( name = "primero" ) String primero,
      @WebParam( name = "segundo" ) String segundo )
   {
      EnteroEnorme operando1 = EnteroEnorme.analizarEnteroEnorme( primero );
      EnteroEnorme operando2 = EnteroEnorme.analizarEnteroEnorme( segundo );
      EnteroEnorme resultado = new EnteroEnorme(); // almacena la diferencia

      // resta el dígito inferior del dígito superior
      for ( int i = 0; i < MAXIMO; i++ )
      {
```

Figura 28.2 | Servicio Web `EnteroEnorme` que realiza operaciones con enteros grandes. (Parte 2 de 3).

```
         // si el dígito en operando1 es menor que el correspondiente
         // dígito en operando2, pide prestado al siguiente dígito
         if ( operando1.numero[ i ] < operando2.numero[ i ] )
            operando1.pedirprestado( i );

            // resta los dígitos
            resultado.numero[ i ] = operando1.numero[ i ] - operando2.numero[ i ];
      } // fin de for

      return resultado.toString();
   } // fin del Método Web restar

   // pide prestado 1 del siguiente dígito
   private void pedirprestado( int lugar )
   {
      if ( lugar >= MAXIMO )
         throw new IndexOutOfBoundsException();
      else if ( numero[ lugar + 1 ] == 0 ) // si el siguiente dígito es cero
         pedirprestado( lugar + 1 ); // pide prestado del siguiente dígito

      numero[ lugar ] += 10; // suma 10 al dígito que prestó
      --numero[ lugar + 1 ]; // resta uno al dígito de la izquierda
   } // fin del método pedirprestado

   // Método Web que devuelve true si el primer entero es mayor que el segundo
   @WebMethod( operationName = "mayor" )
   public boolean mayor( @WebParam( name = "primero" ) String primero,
      @WebParam( name = "segundo" ) String segundo )
   {
      try // trata de restar el primer número del segundo
      {
         String diferencia = restar( primero, segundo );
         return !diferencia.matches( "^[0]+$" );
      } // fin de try
      catch ( IndexOutOfBoundsException e ) // primero es menor que segundo
      {
         return false;
      } // fin de catch
   } // fin del Método Web mayor

   // Método Web que devuelve true si el primer entero es menor que el segundo
   @WebMethod( operationName = "menor" )
   public boolean menor( @WebParam( name = "primero" ) String primero,
      @WebParam( name = "segundo" ) String segundo )
   {
      return mayor( segundo, primero );
   } // fin del Método Web menor

   // Método Web que devuelve true si el primer entero es igual al segundo
   @WebMethod( operationName = "igual" )
   public boolean igual( @WebParam( name = "primero" ) String primero,
      @WebParam( name = "segundo" ) String segundo )
   {
      return !( mayor( primero, segundo ) || menor( primero, segundo ) );
   } // fin del Método Web igual
} // fin de la clase EnteroEnorme
```

Figura 28.2 | Servicio Web `EnteroEnorme` que realiza operaciones con enteros grandes. (Parte 3 de 3).

En las líneas 5 a 7 se importan las anotaciones que se utilizan en este ejemplo. De manera predeterminada, cada nueva clase de servicio Web que se crea con las APIs de JAX-WS es un **POJO (Objeto Java simple)**, lo cual significa que, a diferencia de las APIs de servicios Web de Java anteriores, no necesita extender una clase o implementar una interfaz para crear un servicio Web. Al compilar una clase que utiliza estas anotaciones de JAX-WS 2.0, el compilador crea todos los **artefactos del lado servidor** que soportan el servicio Web; es decir, el marco de trabajo de código compilado que permite al servicio Web esperar las peticiones de los clientes y responder a esas peticiones, una vez que el servicio se despliega en un servidor de aplicaciones. Algunos servidores de aplicaciones populares que soportan los servicios Web de Java son: Sun Java System Application Server (`www.sun.com/software/products/appsrvr/index.xml`), GlassFish (`glassfish.dev.java.net`), Apache Tomcat (`tomcat.apache.org`), BEA Weblogic Server (`www.bea.com`), y JBoss Application Server (`www.jboss.org/products/jbossas`). En este capítulo utilizaremos Sun Java System Application Server.

Las líneas 9 a 11 contienen una anotación `@WebService` (importada en la línea 5) con las propiedades `name` y `serviceName`. La **anotación `@WebService`** indica que la clase `EnteroEnorme` representa a un servicio Web. La anotación va seguida por un conjunto de paréntesis que contienen elementos opcionales. El **elemento `name`** de la anotación (línea 10) especifica el nombre de la clase proxy que se generará para el cliente. El **elemento `serviceName`** (línea 11) especifica el nombre de la clase que el cliente debe utilizar para obtener un objeto de la clase proxy. [*Nota:* si el elemento `serviceName` no se especifica, se asume que el nombre del servicio Web será el nombre de la clase, seguido por la palabra `Service`]. Netbeans coloca la anotación `@WebService` al principio de cada nueva clase de servicio Web que el programador crea. Después se pueden agregar las propiedades `name` y `serviceName` en el paréntesis que sigue a la anotación.

En la línea 14 se declara la constante `MAXIMO`, la cual especifica el número máximo de dígitos para un `EnteroEnorme` (en este ejemplo, 100). En la línea 15 se crea el arreglo que almacena los dígitos en un entero enorme. En las líneas 18 a 40 se declara el método `toString`, el cual devuelve una representación `String` de un `EnteroEnorme`, sin ceros a la izquierda. En las líneas 43 a 52 se declara el método `static` llamado `analizarEnteroEnorme`, el cual convierte un objeto `String` en un objeto `EnteroEnorme`. Los métodos `sumar`, `restar`, `mayor`, `menor` e `igual` del servicio Web utilizan `analizarEnteroEnorme` para convertir sus argumentos `String` en objetos `EnteroEnorme` para procesarlos.

Los métodos `sumar`, `restar`, `mayor`, `menor` e `igual` de `EnteroEnorme` se marcan con la **anotación `@WebMethod`** (líneas 55, 81, 117, 133 y 141) para indicar que pueden llamarse en forma remota. Los métodos que no se marcan con `@WebMethod` no son accesibles para los clientes que consumen el servicio Web. Por lo general, dichos miembros son métodos utilitarios dentro de la clase del servicio Web. Observe que cada una de las anotaciones `@WebMethod` utiliza el elemento **`operationName`** para especificar el nombre del método que está expuesto al cliente del servicio Web.



Error común de programación 28.1

Si no se expone un método como método Web, declarándolo con la anotación `@WebMethod`, los clientes del servicio Web no podrán acceder a ese método.



Error común de programación 28.2

Los métodos con la anotación `@WebMethod` no pueden ser `static`. Debe existir un objeto de la clase de servicio Web para que un cliente pueda acceder a los métodos del servicio Web.

Cada uno de los métodos Web en la clase `EnteroEnorme` especifica parámetros que se anotan con la **anotación `@WebParam`** (por ejemplo, las líneas 56 y 57 del método `sumar`). El elemento **`name`** opcional de `@WebParam` indica el nombre del parámetro que está expuesto a los clientes del servicio Web.

En las líneas 55 a 78 y 81 a 102 se declaran los métodos Web `sumar` y `restar` de `EnteroEnorme`. Para simplificar, vamos a suponer que `sumar` no produce un desbordamiento (es decir, el resultado será de 100 dígitos o menor) y que el primer argumento para `restar` siempre será mayor que el segundo. El método `restar` llama al método `pedirprestado` (líneas 105 a 114) cuando es necesario pedir prestado 1 al siguiente dígito a la izquierda en el primer argumento; es decir, cuando un dígito específico en el operando izquierdo es menor que el dígito correspondiente en el operando derecho. El método `pedirprestado` suma 10 al dígito apropiado y resta 1 al siguiente dígito a la izquierda. Este método utilitario no está diseñado para ser llamado en forma remota, por lo que no se marca con `@WebMethod`.

En las líneas 117 a 130 se declara el método Web `mayor` de `EnteroEnorme`. En la línea 123 se invoca el método `restar` para calcular la diferencia entre los números. Si el primer número es menor que el segundo, se produce una excepción. En este caso, `mayor` devuelve `false`. Si `restar` no lanza una excepción, entonces en la línea 124 se devuelve el resultado de la expresión

```
!diferencia.matches( "^[0]+$" )
```

Esta expresión llama al método `String matches` para determinar si el objeto `String diferencia` coincide con la expresión regular `"^[0]+$"`, la cual determina si el objeto `String` consiste sólo de uno o más ceros. Los símbolos `^` y `$` indican que `matches` debe devolver `true` sólo si todo el objeto `String diferencia` coincide con la expresión regular. Después utilizamos el operador lógico de negación (`!`) para devolver el valor `boolean` opuesto. De esta forma, si los números son iguales (es decir, si su diferencia es 0), la expresión anterior devuelve `false`; el primer número no es mayor que el segundo. En cualquier otro caso, la expresión devuelve `true`. En la sección 30.7 hablaremos con detalle sobre las expresiones regulares.

En las líneas 133 a 146 se declaran los métodos `menor` e `igual`. El método `menor` devuelve el resultado de invocar al método `mayor` (línea 137) con los argumentos invertidos; si `primero` es menor que `segundo`, entonces `segundo` es mayor que `primero`. El método `igual` invoca a los métodos `mayor` y `menor` (línea 145). Si `mayor` o `menor` devuelven `true`, en la línea 145 se devuelve `false` debido a que los números no son iguales. Si ambos métodos devuelven `false`, los números son iguales y en la línea 145 se devuelve `true`.

28.3.3 Publicación del servicio Web `EnteroEnorme` desde Netbeans

Ahora que hemos creado la clase de servicio Web `EnteroEnorme`, utilizaremos Netbeans para generar y publicar (es decir, desplegar) el servicio Web, de manera que los clientes puedan consumir sus servicios. Netbeans se encarga de todos los detalles relacionados con la generación y despliegue de un servicio Web por nosotros. Esto incluye la creación del marco de trabajo requerido para dar soporte al servicio Web. Haga clic con el botón derecho del ratón en el nombre del proyecto (`EnteroEnorme`), en la ficha **Projects** de Netbeans para que aparezca el menú contextual que se muestra en la figura 28.3. Para determinar si hay errores de compilación en el proyecto, seleccione la opción **Build Project**. Cuando el proyecto se compile con éxito, puede seleccionar **Deploy Project** para desplegar el proyecto en el servidor que seleccionó cuando configuró la aplicación Web en la sección 28.3.1. Si el código en el proyecto ha cambiado desde la última vez que se generó, al seleccionar **Deploy Project** también se generará. Al seleccionar **Run Project** se ejecuta la aplicación Web. Si ésta no se había generado o desplegado antes, esta opción ejecuta primero estas tareas. Observe que las opciones **Deploy Project** y **Run Project** también inician

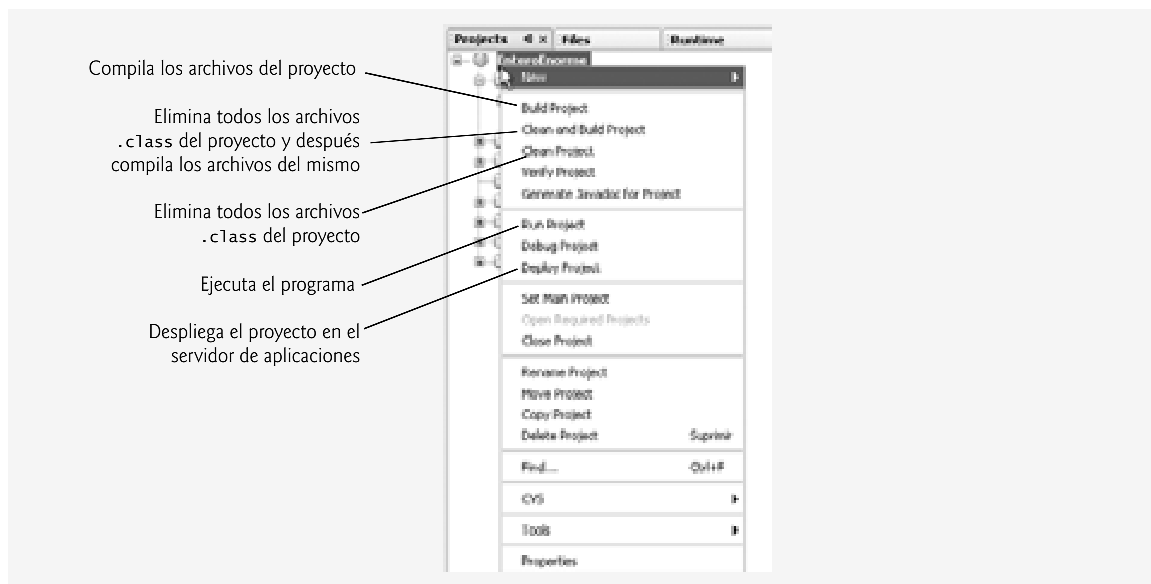


Figura 28.3 | Menú contextual que aparece al hacer clic con el botón derecho del ratón en el nombre de un proyecto, en la ficha **Projects** de Netbeans.

el servidor de aplicación (en nuestro caso, Sun Java System Application Server), en caso de que no se encuentre ya en ejecución. Para asegurar que todos los archivos de código fuente en un proyecto se vuelvan a compilar durante la siguiente operación de generación, podemos usar las opciones **Clean Project** y **Build Project**. Si no lo ha hecho todavía, seleccione **Deploy Project** ahora.

28.3.4 Prueba del servicio Web `EnteroEnorme` con la página Web `Tester` de Sun Java System Application Server

El siguiente paso es probar el servicio Web `EnteroEnorme`. Anteriormente seleccionamos el servidor Sun Java System Application Server para ejecutar la aplicación Web. Este servidor puede crear una página Web en forma dinámica, para probar los métodos de un servicio Web desde un navegador Web. Para habilitar esta capacidad:

1. Haga clic con el botón derecho del ratón en el nombre del proyecto (`EnteroEnorme`) en la ficha **Projects** de Netbeans, y seleccione **Properties** en el menú contextual para que aparezca el cuadro de diálogo **Project Properties**.
2. Haga clic en la opción **Run** de **Categories** para mostrar las opciones para ejecutar el proyecto.
3. En el campo **Relative URL**, escriba `/HugeIntegerService?Tester`.
4. Haga clic en **OK** para cerrar el cuadro de diálogo **Project Properties**.

El campo **Relative URL** especifica lo que debe ocurrir cuando se ejecute la aplicación Web. Si este campo está vacío, entonces se mostrará la JSP predeterminada de la aplicación Web al ejecutar el proyecto. Si especificamos `/EnteroEnormeService?Tester` en este campo y después ejecutamos el proyecto, Sun Java System Application Server genera la página Web `Tester` y la carga en el navegador Web. En la figura 28.4 se muestra la página Web `Tester` para el servicio Web `EnteroEnorme`. Una vez que haya desplegado el servicio Web, también puede escribir el URL

```
http://localhost:8080/EnteroEnorme/EnteroEnormeService?Tester
```



Figura 28.4 | Página Web `Tester` creada por Sun Java System Application Server para el servicio Web `EnteroEnorme`.

en su navegador Web para ver la página Web `Tester`. Observe que `ServicioEnteroEnorme` es el nombre (especificado en la línea 11 de la figura 28.2) que los clientes, incluyendo la página Web `Tester`, utilizan para acceder al servicio Web.

Para probar los métodos Web de `EnteroEnorme`, escriba dos enteros positivos en los campos de texto a la derecha del botón de un método específico, y después haga clic en el botón para invocar el método Web y ver el resultado. En la figura 28.5 se muestran los resultados de invocar al método `sumar` de `EnteroEnorme` con los valores `99999999999999999` y `1`. Observe que el número `99999999999999999` es más grande de lo que el tipo primitivo `long` puede representar.

a) Invocación del método `sumar` del servicio Web `EnteroEnorme`.

b) Resultados de llamar al método `sumar` del servicio Web `EnteroEnorme` con `"99999999999999999"` y `"1"`.

Figura 28.5 | Prueba del método `sumar` de `EnteroEnorme`.

Observe que puede acceder al servicio Web sólo cuando el servidor de aplicaciones esté en ejecución. Si Netbeans lanza el servidor de aplicaciones por usted, lo apagará de manera automática cuando cierre Netbeans. Para mantener el servidor de aplicaciones funcionando, puede iniciarlo de manera independiente a Netbeans antes de desplegar aplicaciones Web en Netbeans. Para Sun Java System Application Server ejecutándose en Windows, puede hacer esto seleccionando **Inicio > Todos los programas > Sun Microsystems > Application Server PE 9 > Start Default Server**. Para cerrar el servidor de aplicaciones, seleccione la opción **Stop Default Server** de la misma ubicación.

Prueba del servicio Web `EnteroEnorme` desde otra computadora

Si su computadora está conectada a una red y permite peticiones HTTP, entonces puede probar el servicio Web desde otra computadora en la red, escribiendo el siguiente URL en un navegador en otra computadora:

```
http://host:8080/EnteroEnorme/ServicioEnteroEnorme?Tester
```

en donde *host* es el nombre de host o dirección IP de la computadora en la que está desplegado el servicio Web.

Nota para los usuarios de Windows XP Service Pack 2

Por cuestiones de seguridad, las computadoras que ejecutan Windows XP Service Pack 2 no permiten peticiones HTTP de otras computadoras de manera predeterminada. Si desea que otras computadoras se conecten a su computadora mediante HTTP, realice los siguientes pasos:

1. Seleccione **Inicio > Panel de control** para abrir la ventana **Panel de control** de su sistema, y después haga doble clic en **Firewall de Windows** para ver el cuadro de diálogo de configuración del **Firewall de Windows**.
2. En este cuadro de diálogo de configuración, haga clic en la ficha **Opciones avanzadas**, seleccione **Conexión de área local** (o el nombre de su conexión de red, si es distinto) en el cuadro de lista **Configuración de conexión de red**, y haga clic en el botón **Configuración…** para que aparezca el cuadro de diálogo **Configuración avanzada**.
3. En el cuadro de diálogo **Configuración avanzada**, asegúrese de que la casilla de verificación para **Servidor Web (HTTP)** esté seleccionada, para permitir que los clientes en otras computadoras puedan enviar peticiones al servidor Web de su computadora.
4. Haga clic en **Aceptar** en el cuadro de diálogo **Configuración avanzada** y después en **Aceptar** en el cuadro de diálogo de configuración de **Firewall de Windows**.

28.3.5 Descripción de un servicio Web con el Lenguaje de descripción de servicios Web (WSDL)

Una vez que se implementa un servicio Web, se compila y se despliega en un servidor de aplicaciones, una aplicación cliente puede consumirlo. Sin embargo, para ello el cliente debe saber en dónde encontrar el servicio Web, y se le debe proporcionar una descripción de cómo interactuar con el servicio Web; es decir, qué métodos están disponibles, qué parámetros esperan y qué devuelve cada método. Para este fin, JAX-WS utiliza el **Lenguaje de descripción de servicios Web (WSDL)**: un vocabulario XML estándar para describir los servicios Web en forma independiente de la plataforma.

No es necesario comprender los detalles de WSDL para aprovechar sus beneficios; el servidor genera un WSDL del servicio Web en forma dinámica por usted, y las herramientas cliente pueden analizar el WSDL para ayudar a crear la clase proxy del lado cliente que un cliente utiliza para acceder al servicio Web. Como el WSDL se crea en forma dinámica, los clientes siempre reciben la descripción más actualizada de un servicio Web desplegado. Para ver el WSDL del servicio Web `EnteroEnorme` (figura 28.6), escriba el siguiente URL en su navegador:

```
http://localhost:8080/EnteroEnorme/ServicioEnteroEnorme?WSDL
```

o haga clic en el vínculo **WSDL File** en la página Web `Tester` (que se muestra en la figura 28.4).

Cómo acceder al WSDL del servicio Web `EnteroEnorme` desde otra computadora

En algún momento dado, los clientes en otras computadoras querrán utilizar su servicio Web. Dichos clientes necesitan acceso al WSDL de su servicio Web, al cual pueden acceder mediante el siguiente URL:

`http://`*host*`:8080/EnteroEnorme/ServicioEnteroEnorme?WSDL`

en donde *host* es el nombre de host o dirección IP de la computadora en la que se va a desplegar el servicio Web. Como vimos en la sección 28.3.4, esto funcionará sólo si su computadora permite conexiones HTTP desde otras computadoras, como se da el caso para los servidores Web y de aplicaciones que están disponibles públicamente.

28.4 Cómo consumir un servicio Web

Ahora que hemos definido y desplegado nuestro servicio Web, podemos consumirlo desde una aplicación cliente. El cliente de un servicio Web puede ser cualquier tipo de aplicación, o incluso otro servicio Web. Para habilitar una aplicación cliente, de manera que pueda consumir un servicio Web, hay que **agregar una referencia del servicio Web** a la aplicación. Este proceso define la clase proxy que permite al cliente acceder al servicio Web.

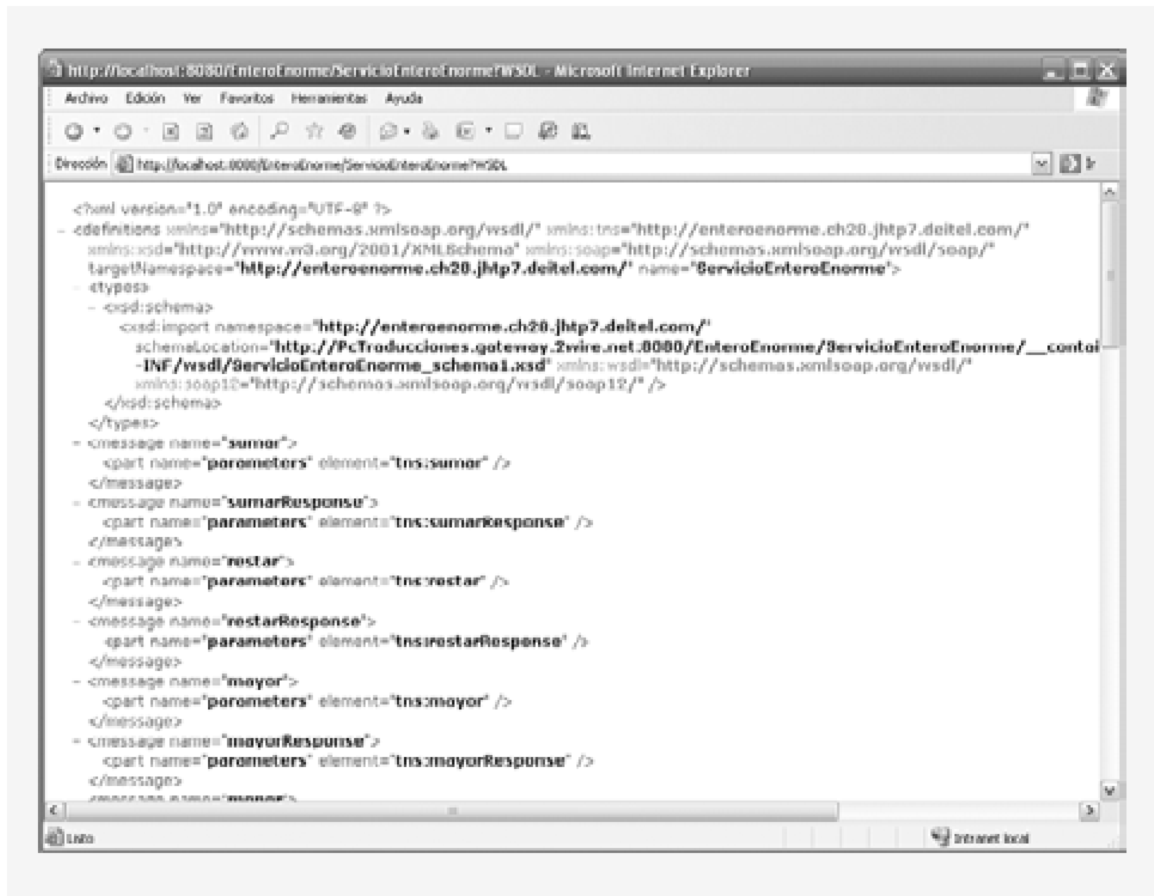


Figura 28.6 | Un fragmento del archivo `.wsdl` para el servicio Web `EnteroEnorme`.

28.4.1 Creación de un cliente para consumir el servicio Web `EnteroEnorme`

En esta sección, utilizará Netbeans para crear una aplicación Java de GUI de escritorio, y después agregará una referencia de servicio Web al proyecto para que el cliente pueda acceder al servicio Web. Al agregar la referencia de servicio Web, el IDE crea y compila los **artefactos del lado cliente**: el marco de trabajo de código de Java que da soporte a la clase proxy del lado cliente. Después, el cliente llama a los métodos en un objeto de la clase proxy, la cual utiliza el resto de los artefactos para interactuar con el servicio Web.

Creación de un proyecto de aplicación de escritorio en Netbeans 5.5

Antes de realizar los pasos en esta sección, asegúrese de que el servicio Web `EnteroEnorme` se haya desplegado, y de que Sun Java System Application Server se esté ejecutando (vea la sección 28.3.3). Realice los siguientes pasos para crear una aplicación de escritorio Java cliente en Netbeans:

1. Seleccione **File > New Project...** para abrir el cuadro de diálogo **New Project**.
2. Seleccione **General** de la lista **Categories** y **Java Application** de la lista **Projects**.
3. Especifique el nombre `UsoEnteroEnorme` en el campo **Project Name** y desactive la casilla de verificación **Create Main Class**. En un momento agregará una subclase de `JFrame` que contiene un método `main`.
4. Haga clic en **Finish** para crear el proyecto.

Cómo agregar una referencia de servicio Web a una aplicación

A continuación, agregará una referencia de servicio Web a su aplicación para que pueda interactuar con el servicio Web `EnteroEnorme`. Para agregar una referencia de servicio Web, realice los siguientes pasos.

1. Haga clic con el botón derecho del ratón en el nombre del proyecto (`UsoEnteroEnorme`) en la ficha **Projects** de Netbeans.
2. Seleccione **New > Web Service Client...** del menú contextual para mostrar el cuadro de diálogo **New Web Service Client** (figura 28.7).
3. En el campo **WSDL URL**, especifique el URL `http://localhost:8080/EnteroEnorme/ServicioEnteroEnorme?WSDL` (figura 28.7). Este URL indica al IDE en dónde puede encontrar la descripción WSDL del servicio Web. [*Nota:* si Sun Java System Application Server está ubicado en otro equipo, sustituya `localhost` con el nombre de host o dirección IP de esa computadora]. El IDE utiliza esta descripción WSDL para generar los artefactos del lado cliente que componen y dan soporte al proxy. Observe que el cuadro de diálogo **New Web Service Client** le permite buscar servicios Web en varias ubicaciones. Muchas compañías simplemente distribuyen los URL de WSDL exactos para sus servicios Web, que el programador puede colocar en el campo **WSDL URL**.

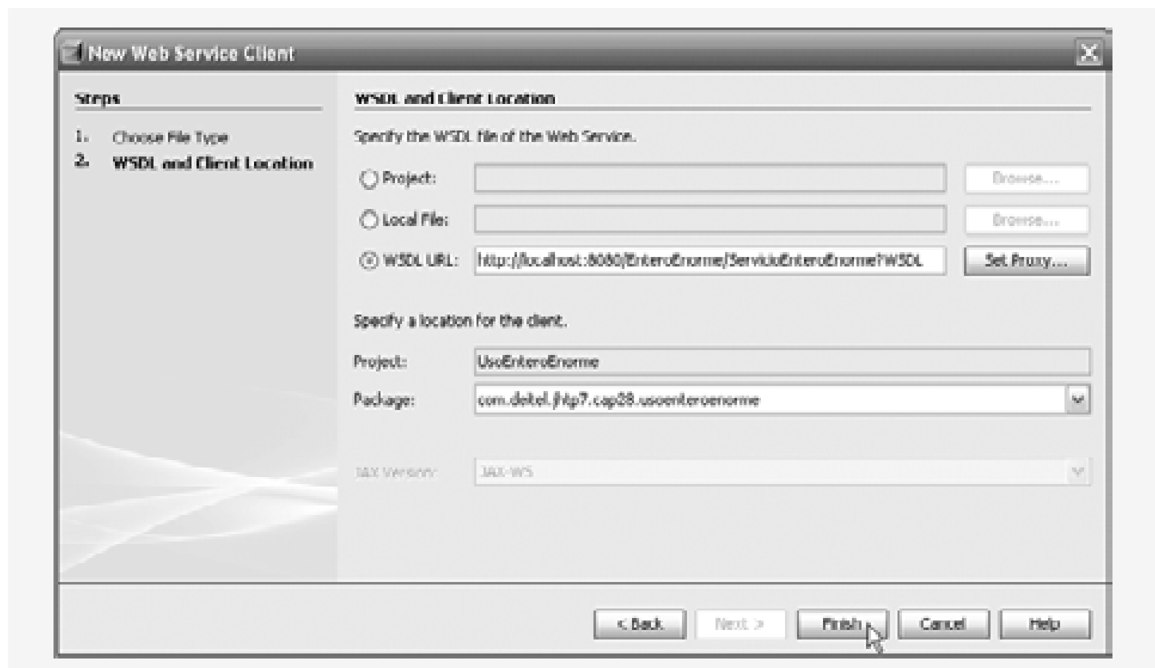


Figura 28.7 | Cuadro de diálogo **New Web Service Client.**

4. En el campo **Package**, especifique `com.deitel.jhtp7.cap28.usoenteroenorme` como el nombre del paquete.
5. Haga clic en **Finish** para cerrar el cuadro de diálogo **New Web Service Client**.

En la ficha **Projects** de Netbeans, el proyecto `UsoEnteroEnorme` ahora contiene una carpeta `Web Service References` con el proxy para el servicio Web `EnteroEnorme` (figura 28.8). Observe que el nombre del proxy se lista como `ServicioEnteroEnorme`, como especificamos en la línea 11 de la figura 28.2.

Al especificar el servicio Web que deseamos consumir, Netbeans accede a la información WSDL del servicio Web y la copia en un archivo en nuestro proyecto (llamado `ServicioEnteroEnorme.wsdl` en este ejemplo). Para

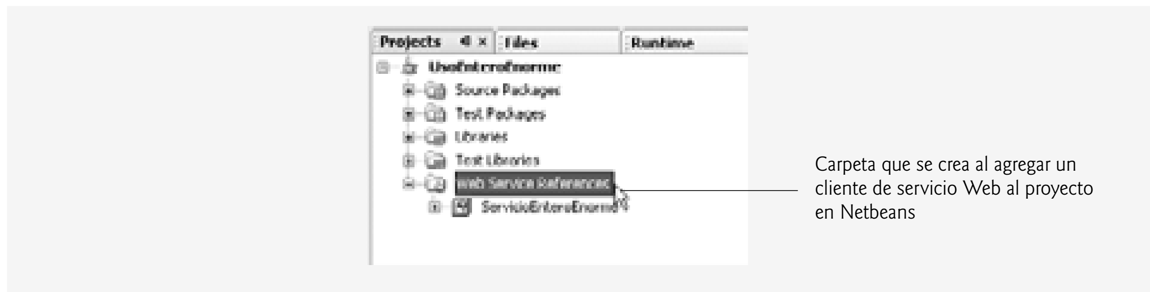


Figura 28.8 | Ficha **Project** de Netbeans después de agregar una referencia de servicio Web al proyecto.

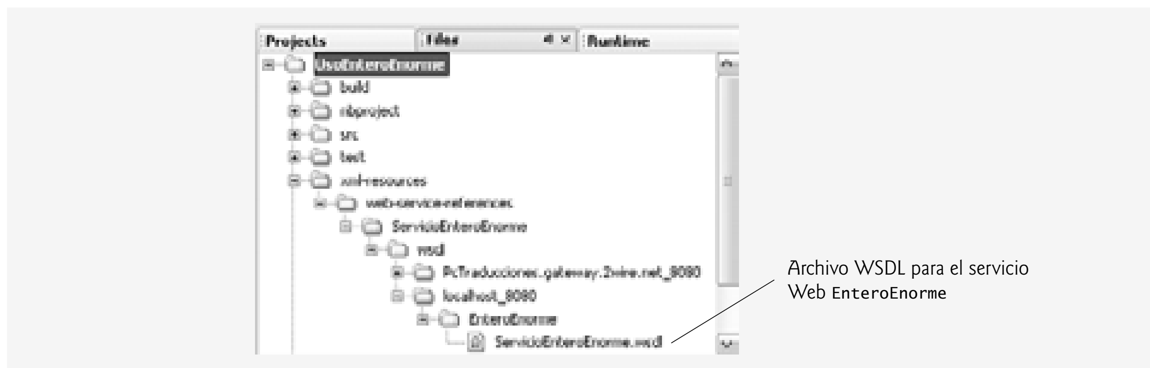


Figura 28.9 | Ubicación del archivo `ServicioEnteroEnorme.wsdl` en la ficha **Files** de Netbeans.

ver este archivo desde la ficha **Files** de Netbeans, expanda los nodos en la carpeta `xml-resources` del proyecto `UsoEnteroEnorme` como se muestra en la figura 28.9. Si el servicio Web cambia, los artefactos del lado cliente y la copia del archivo WSDL del cliente se pueden regenerar haciendo clic con el botón derecho del ratón en el nodo `ServicioEnteroEnorme` que se muestra en la figura 28.8, y seleccionando **Refresh Client**.

Para ver los artefactos del lado cliente generados por el IDE seleccione la ficha **Files** de Netbeans y expanda la carpeta **build** del proyecto `UsoEnteroEnorme`, como se muestra en la figura 28.10.

28.4.2 Cómo consumir el servicio Web `EnteroEnorme`

Para este ejemplo, utilizaremos una aplicación de GUI para interactuar con el servicio Web. Para crear la GUI de la aplicación cliente, primero hay que agregar una subclase de `JFrame` al proyecto. Para ello, realice los siguientes pasos:

1. Haga clic con el botón derecho del ratón en el nombre del proyecto en la ficha **Project** de Netbeans.
2. Seleccione **New > JFrame Form...** para que aparezca el cuadro de diálogo **New JFrame Form**.
3. Especifique `UsoEnteroEnormeJFrame` en el campo **Class Name**.
4. Especifique `com.deitel.jhtp7.cap28.clienteenteroenorme` en el campo **Package**.
5. Haga clic en **Finish** para cerrar el cuadro de diálogo **New JFrame Form**.

A continuación, cree la GUI que se muestra en las capturas de pantalla de ejemplo al final de la figura 28.11. Para obtener más información acerca de cómo usar Netbeans en la creación de una GUI y de manejadores de eventos, consulte el apéndice de `GroupLayout`.

La aplicación en la figura 28.11 utiliza el servicio Web `EnteroEnorme` para realizar cálculos con enteros positivos de hasta `100` dígitos. Para ahorrar espacio, no mostramos el método `initComponents` generado en forma automática por Netbeans, el cual contiene el código que crea los componentes de GUI, los posiciona y registra sus

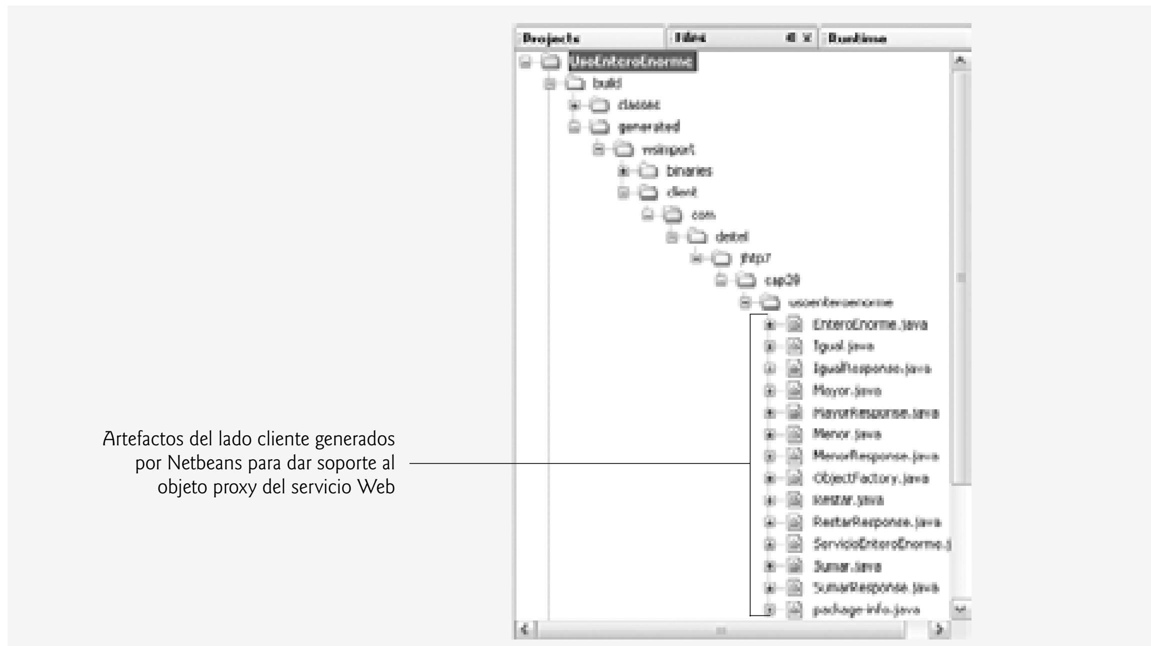


Figura 28.10 | Cómo ver los artefactos del lado cliente del servicio Web `EnteroEnorme` generados por Netbeans.

manejadores de eventos. Para ver el código fuente completo, abra el archivo `UsoEnteroEnormeJFrame.java` que se encuentra en la carpeta de este ejemplo, en `src\java\com\deitel\jhtp7\cap28\clienteenteroenorme`. Observe que Netbeans coloca las declaraciones de variables de instancia de los componentes de GUI al final de la clase (líneas 326 a 335). Java permite declarar variables de instancia en cualquier parte del cuerpo de una clase, siempre y cuando se coloquen fuera de los métodos de la clase. Seguiremos declarando nuestras propias variables de instancia en la parte superior de la clase.

```
1  // Fig. 28.11: UsoEnteroEnormeJFrame.java
2  // Aplicación de escritorio cliente para el servicio Web EnteroEnorme.
3  package com.deitel.jhtp7.cap28.clienteenteroenorme;
4
5  // importa las clases para acceder al proxy del servicio Web EnteroEnorme
6  import com.deitel.jhtp7.cap28.clienteenteroenorme.EnteroEnorme;
7  import com.deitel.jhtp7.cap28.clienteenteroenorme.ServicioEnteroEnorme;
8
9  import javax.swing.JOptionPane; // se utiliza para mostrar errores al usuario
10
11 public class UsoEnteroEnormeJFrame extends javax.swing.JFrame
12 {
13    private ServicioEnteroEnorme servicioEnteroEnorme; // se usa para obtener el proxy
14    private EnteroEnorme proxyEnteroEnorme; // se usa para acceder al servicio Web
15
16    // constructor sin argumentos
17    public UsoEnteroEnormeJFrame()
18    {
19       initComponents();
20
```

Figura 28.11 | Aplicación de escritorio cliente para el servicio Web `EnteroEnorme`. (Parte 1 de 6).

```
      try
      {
         // crea los objetos para acceder al servicio Web EnteroEnorme
         servicioEnteroEnorme = new ServicioEnteroEnorme();
         proxyEnteroEnorme = servicioEnteroEnorme.getEnteroEnormePort();
      }
      catch ( Exception excepcion )
      {
         excepcion.printStackTrace();
      }
   } // fin del constructor de UsoEnteroEnormeJFrame

   // El método initComponents se genera automáticamente por Netbeans y se llama
   // desde el constructor para inicializar la GUI. No mostraremos aquí este método
   // para ahorrar espacio. Abra UsoEnteroEnormeJFrame.java en la carpeta
   // de este ejemplo para ver el código generado completo (líneas 37 a 153).

   // invoca al método sumar del servicio Web EnteroEnorme para sumar objetos EnteroEnorme
   private void sumarJButtonActionPerformed(
      java.awt.event.ActionEvent evt )
   {
      String primerNumero = primeroJTextField.getText();
      String segundoNumero = segundoJTextField.getText();

      if ( esValido( primerNumero ) && esValido( segundoNumero ) )
      {
         try
         {
            resultadosJTextArea.setText(
               proxyEnteroEnorme.sumar( primerNumero, segundoNumero ) );
         } // fin de try
         catch ( Exception e )
         {
            JOptionPane.showMessageDialog( this, e.toString(),
               "Error en el metodo Sumar", JOptionPane.ERROR_MESSAGE );
            e.printStackTrace();
         } // fin de catch
      } // fin de if
   } // fin del método sumarJButtonActionPerformed

   // invoca al método restar del servicio Web EnteroEnorme para restar el
   // segundo EnteroEnorme del primero
   private void restarJButtonActionPerformed(
      java.awt.event.ActionEvent evt )
   {
      String primerNumero = primeroJTextField.getText();
      String segundoNumero = segundoJTextField.getText();

      if ( esValido( primerNumero ) && esValido( segundoNumero ) )
      {
         try
         {
            resultadosJTextArea.setText(
               proxyEnteroEnorme.restar( primerNumero, segundoNumero ) );
         } // fin de try
         catch ( Exception e )
         {
            JOptionPane.showMessageDialog( this, e.toString(),
```

Figura 28.11 | Aplicación de escritorio cliente para el servicio Web `EnteroEnorme`. (Parte 2 de 6).

```
                "Error en el metodo Restar", JOptionPane.ERROR_MESSAGE );
            e.printStackTrace();
         } // fin de catch
      } // fin de if
   } // fin del método restarJButtonActionPerformed

   // invoca al método mayor del servicio Web EnteroEnorme para determinar si
   // el primer EnteroEnorme es mayor que el segundo
   private void mayorJButtonActionPerformed(
      java.awt.event.ActionEvent evt )
   {
      String primerNumero = primeroJTextField.getText();
      String segundoNumero = segundoJTextField.getText();

      if ( esValido( primerNumero ) && esValido( segundoNumero ) )
      {
         try
         {
            boolean result =
               proxyEnteroEnorme.mayor( primerNumero, segundoNumero );
            resultadosJTextArea.setText( String.format( "%s %s %s %s",
               primerNumero, ( result ? "es" : "no es" ), "mayor que",
               segundoNumero ) );
         } // fin de try
         catch ( Exception e )
         {
            JOptionPane.showMessageDialog( this, e.toString(),
               "Error en metodo Mayor", JOptionPane.ERROR_MESSAGE );
            e.printStackTrace();
         } // fin de catch
      } // fin de if
   } // fin del método mayorJButtonActionPerformed

   // invoca al método menor del servicio Web EnteroEnorme para determinar
   // si el primer EnteroEnorme es menor que el segundo
   private void menorJButtonActionPerformed(
      java.awt.event.ActionEvent evt )
   {
      String primerNumero = primeroJTextField.getText();
      String segundoNumero = segundoJTextField.getText();

      if ( esValido( primerNumero ) && esValido( segundoNumero ) )
      {
         try
         {
            boolean resultado =
               proxyEnteroEnorme.menor( primerNumero, segundoNumero );
            resultadosJTextArea.setText( String.format( "%s %s %s %s",
               primerNumero, ( resultado ? "es" : "no es" ), "menor que",
               segundoNumero ) );
         } // fin de try
         catch ( Exception e )
         {
            JOptionPane.showMessageDialog( this, e.toString(),
               "Error en metodo Menor", JOptionPane.ERROR_MESSAGE );
            e.printStackTrace();
         } // fin de catch
      } // fin de if
```

Figura 28.11 | Aplicación de escritorio cliente para el servicio Web `EnteroEnorme`. (Parte 3 de 6).

```
   } // fin del método menorJButtonActionPerformed

   // invoca al método igual del servicio Web EnteroEnorme para determinar si
   // el primer EnteroEnorme es igual al segundo
   private void igualJButtonActionPerformed(
      java.awt.event.ActionEvent evt )
   {
      String primerNumero = primeroJTextField.getText();
      String segundoNumero = segundoJTextField.getText();

      if ( esValido( primerNumero ) && esValido( segundoNumero ) )
      {
         try
         {
            boolean resultado =
               proxyEnteroEnorme.igual( primerNumero, segundoNumero );
            resultadosJTextArea.setText( String.format( "%s %s %s %s",
               primerNumero, ( resultado ? "es" : "no es" ), "igual a",
               segundoNumero ) );
         } // fin de try
         catch ( Exception e )
         {
            JOptionPane.showMessageDialog( this, e.toString(),
               "Error en el metodo Igual", JOptionPane.ERROR_MESSAGE );
            e.printStackTrace();
         } // fin de catch
      } // fin de if
   } // fin del método igualJButtonActionPerformed

   // comprueba el tamaño de un objeto String para asegurar que no sea demasiado grande
   // como para usarlo como un EnteroEnorme; asegura que sólo haya dígitos en el String
   private boolean esValido( String numero )
   {
      // comprueba la longitud del objeto String
      if ( numero.length() > 100 )
      {
         JOptionPane.showMessageDialog( this,
            "Los EnterosEnormes deben ser <= 100 digitos.", "Desbordamiento de
            EnteroEnorme",
            JOptionPane.ERROR_MESSAGE );
         return false;
      } // fin de if

      // busca caracteres que no sean dígitos en el objeto String
      for ( char c : numero.toCharArray() )
      {
         if ( !Character.isDigit( c ) )
         {
            JOptionPane.showMessageDialog( this,
               "Hay caracteres que no son digitos en el objeto String",
               "EnteroEnorme contiene caracteres que no son digitos",
               JOptionPane.ERROR_MESSAGE );
            return false;
         } // fin de if
      } // fin de for

      return true; // el número se puede usar como un EnteroEnorme
   } // fin del método de validación
```

Figura 28.11 | Aplicación de escritorio cliente para el servicio Web `EnteroEnorme`. (Parte 4 de 6).

```

   // el método main empieza la ejecución
   public static void main( String args[] )
   {
      java.awt.EventQueue.invokeLater(
         new Runnable()
         {
            public void run()
            {
               new UsoEnteroEnormeJFrame().setVisible( true );
            } // fin del método run
         } // fin de la clase interna anónima
      ); // fin de la llamada a java.awt.EventQueue.invokeLater
   } // fin del método main

   // Variables declaration - do not modify
   private javax.swing.JButton igualJButton;
   private javax.swing.JLabel indicacionesJLabel;
   private javax.swing.JButton mayorJButton;
   private javax.swing.JButton menorJButton;
   private javax.swing.JTextField primeroJTextField;
   private javax.swing.JButton restarJButton;
   private javax.swing.JScrollPane resultadosJScrollPane;
   private javax.swing.JTextArea resultadosJTextArea;
   private javax.swing.JTextField segundoJTextField;
   private javax.swing.JButton sumarJButton;
   // fin de las variables declaration
} // fin de la clase UsoEnteroEnormeJFrame
```



Figura 28.11 | Aplicación de escritorio cliente para el servicio Web `EnteroEnorme`. (Parte 5 de 6).

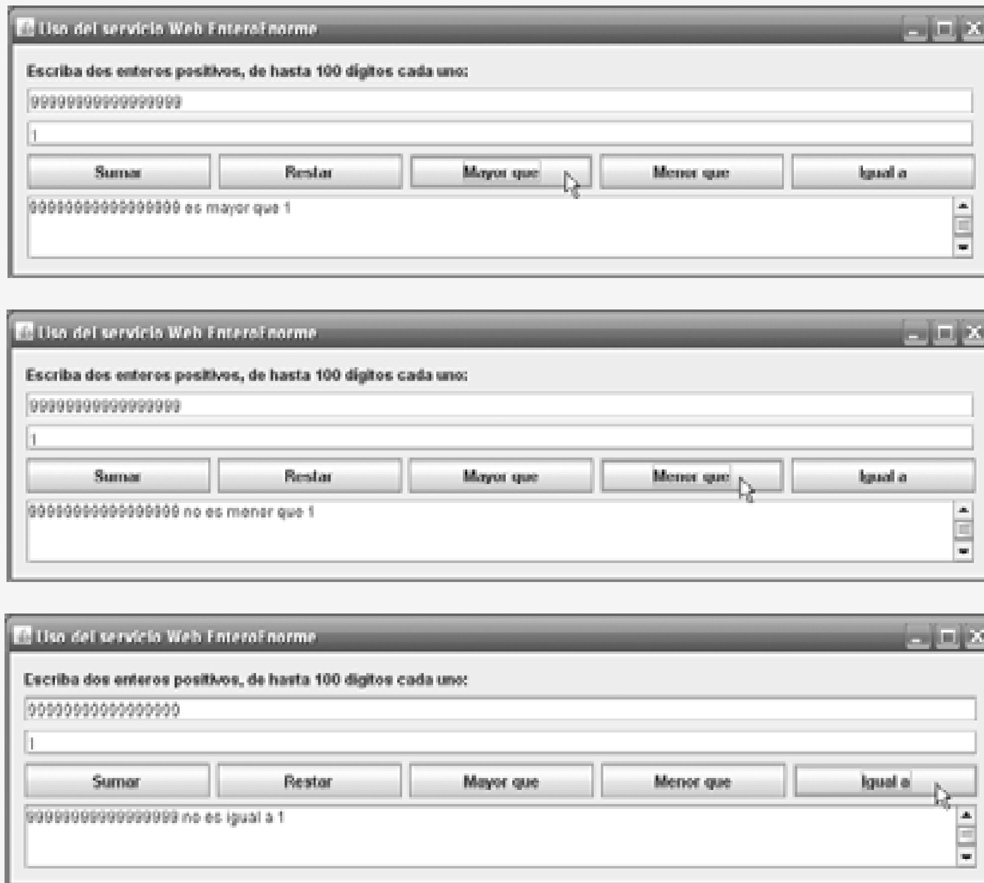


Figura 28.11 | Aplicación de escritorio cliente para el servicio Web `EnteroEnorme`. (Parte 6 de 6).

En las líneas 6 a 7 se importan las clases `EnteroEnorme` y `ServicioEnteroEnorme` que permiten a la aplicación cliente interactuar con el servicio Web. Aquí incluimos esas declaraciones `import` sólo para fines de documentación. Estas clases se encuentran en el mismo paquete que `UsoEnteroEnormeJFrame`, por lo que no son necesarias estas declaraciones `import`. Observe que no tenemos declaraciones `import` para la mayoría de los componentes de GUI que utilizamos en este ejemplo. Al crear una GUI en Netbeans, utiliza los nombres de clases completamente calificados (como `javax.swing.JFrame` en la línea 11), por lo que estas declaraciones no son necesarias.

En las líneas 13 y 14 se declaran las variables de tipo `ServicioEnteroEnorme` y `EnteroEnorme`, respectivamente. En la línea 24 en el constructor se crea un objeto `ServicioEnteroEnorme`. En la línea 25 se utiliza el método `getEnteroEnormePort` de este objeto para obtener el objeto `proxyEnteroEnorme` que utiliza la aplicación para invocar al método del servicio Web.

En las líneas 165 a 166, 189 a 190, 213 a 214, 240 a 241 y 267 a 268 en los diversos manejadores de `JButton`, se invocan los métodos Web del servicio `EnteroEnorme`. Observe que cada llamada se realiza en el objeto proxy local al que se hace referencia mediante `proxyEnteroEnorme`. Después, el objeto proxy se comunica con el servicio Web por el cliente.

El usuario introduce dos enteros, cada uno de hasta 100 dígitos máximo. Al hacer clic en cualquiera de los cinco objetos `JButton`, la aplicación invoca a un método Web para realizar la tarea correspondiente y devolver el resultado. Nuestra aplicación cliente no puede procesar números de 100 dígitos directamente. En vez de ello, el cliente pasa representaciones `String` de esos números a los métodos Web del servicio Web, los cuales realizan

tareas para el cliente. Así, la aplicación cliente utiliza entonces el valor de retorno de cada operación para mostrar un mensaje apropiado.

28.5 SOAP

SOAP (un acrónimo para Protocolo simple de acceso a objetos) es un protocolo independiente de la plataforma que usa XML para facilitar las llamadas a procedimientos remotos, por lo general, a través de HTTP. SOAP es un protocolo común para pasar información entre los clientes de servicios Web y los servicios Web. El protocolo que transmite mensajes de petición y respuesta se conoce también como el **formato de cable** o **protocolo de cable** del servicio Web, ya que define cómo se envía la información "a través del cable".

Cada petición y respuesta se empaqueta en un **mensaje SOAP** (también conocido como **envoltura SOAP**): una "envoltura" XML que contiene la información que un servicio Web requiere para procesar el mensaje. Con unas cuantas excepciones, la mayoría de los **firewalls** (barreras de seguridad que restringen la comunicación entre redes) permiten que pase el tráfico http para que los clientes puedan navegar en sitios Web ubicados en servidores Web detrás de los firewalls. Por ende, XML y HTTP permiten que computadoras en distintas plataformas envíen y reciban mensajes SOAP, con unas cuantas limitaciones.

Los servicios Web también utilizan SOAP para el extenso conjunto de tipos que soporta. El formato de cable utilizado para transmitir peticiones y respuestas debe soportar todos los tipos que se pasan entre las aplicaciones. SOAP soporta tipos primitivos (como `int`) y sus tipos de envoltura (como `Integer`), así como `Date`, `Time` y otros. SOAP también puede transmitir arreglos y objetos de tipos definidos por el usuario (como veremos en la sección 28.8). Para obtener más información acerca de SOAP, visite `www.w3.org/TR/soap/`.

Cuando un programa invoca a un método Web, la petición y toda la información relevante se empaqueta en un mensaje SOAP, y se envía al servidor en el que reside el servicio Web. Este servicio procesa el contenido del mensaje SOAP (que se encuentra dentro de una envoltura SOAP), el cual especifica el método que el cliente desea invocar y los argumentos para este método. A este proceso de interpretar el contenido de un mensaje SOAP se le conoce como **análisis de un mensaje SOAP**. Una vez que el servicio Web recibe y analiza una petición, se hace una llamada al método apropiado con cualquier argumento especificado, y la respuesta se envía de vuelta al cliente en otro mensaje SOAP. El proxy del lado cliente analiza la respuesta, que contiene el resultado de la llamada al método, y devuelve el resultado a la aplicación cliente.

En la figura 28.5 se utilizó la página Web `Tester` del servicio Web `EnteroEnorme` para mostrar el resultado de invocar al método `sumar` de `EnteroEnorme` con los valores `99999999999999999` y `1`. La página Web `Tester` también muestra la petición SOAP y los mensajes de respuesta (que no se mostraron antes). En la figura 28.12 se muestra el mismo resultado con los mensajes SOAP que muestra la aplicación `Tester`. En el mensaje de petición de la figura 28.12, el texto

```
<ns1:sumar>
<primero>99999999999999999</primero>
<segundo>1</segundo>
</ns1:sumar>
```

especifica el método a llamar (`sumar`), los argumentos del método (`primero` y `segundo`), y los valores de los argumentos (`99999999999999999` y `1`). De manera similar, el texto

```
<ns1:sumarResponse>
<return>100000000000000000</return>
</ns1:sumarResponse>
```

del mensaje de respuesta en la figura 28.12 especifica el valor de retorno del método `sumar`.

Al igual que con el WSDL para un servicio Web, los mensajes SOAP se generan de manera automática para usted. Por lo tanto, no necesita comprender los detalles acerca de SOAP o XML para aprovecharlos a la hora de publicar y consumir los servicios Web.

28.6 Rastreo de sesiones en los servicios Web

En la sección 26.7 describimos las ventajas en cuanto al uso del rastreo de sesiones para mantener la información de estado de un cliente, de manera que podamos personalizar las experiencias de navegación del usuario. Ahora vamos a incorporar el rastreo de sesiones en un servicio Web. Suponga que una aplicación cliente necesita llamar a

Figura 28.12 | Mensajes SOAP para el método `sumar` del servicio Web `EnteroEnorme`, como se muestra en la página Web `Tester` de Sun Java System Application Server.

varios métodos del mismo servicio Web, tal vez varias veces cada uno de ellos. En tal caso, puede ser benéfico para el servicio Web mantener la información de estado para el cliente, con lo cual se elimina la necesidad de pasar la información del cliente entre éste y el servicio Web varias veces. Por ejemplo, un servicio Web que proporciona reseñas de restaurantes locales podría almacenar el domicilio del usuario cliente durante la petición inicial, y después utilizarlo para devolver resultados personalizados y localizados en las peticiones subsiguientes. Al almacenar la información de la sesión, también permitimos que un servicio Web diferencie a un cliente de otro.

28.6.1 Creación de un servicio Web `Blackjack`

Nuestro siguiente ejemplo es un servicio Web que le ayudará a desarrollar un juego de cartas llamado Blackjack. El servicio Web `Blackjack` (figura 28.13) proporciona métodos Web para barajar un mazo de cartas, repartir una carta del mazo y evaluar una mano de cartas. Después de presentar el servicio Web, lo utilizaremos para que sirva como el repartidor en un juego de blackjack (figura 28.14). El servicio Web `Blackjack` usa un objeto

`HttpSession` para mantener un mazo único de cartas para cada aplicación cliente. Varios clientes pueden usar el servicio al mismo tiempo, pero las llamadas a los métodos Web que realiza un cliente específico sólo utilizan el mazo de cartas almacenadas en la sesión de ese cliente. Nuestro ejemplo utiliza las siguientes reglas de blackjack:

> *Se reparten dos cartas al repartidor y dos cartas al jugador. Las cartas del jugador se reparten con la cara hacia arriba. Sólo la primera de las cartas del repartidor se reparte con la cara hacia arriba. Cada carta tiene un valor. Una carta numerada del 2 al 10 vale lo que indique en su cara. Los jotos, reinas y reyes valen 10 cada uno. Los ases pueden valer 1 u 11, lo que sea más conveniente para el jugador (como pronto veremos). Si la suma de las dos cartas iniciales del jugador es 21 (por ejemplo, que se hayan repartido al jugador una carta con valor de 10 y un as, que cuenta como 11 en esta ocasión), el jugador tiene "blackjack" y gana el juego de inmediato (si el repartidor tampoco tienen blackjack, lo que resulta en un "empate"). En caso contrario, el jugador puede empezar a tomar cartas adicionales, una a la vez. Estas cartas se reparten con la cara hacia arriba, y el jugador decide cuándo dejar de tomar cartas. Si el jugador "se pasa" (es decir, si la suma de las cartas del jugador excede a 21), el juego termina y el jugador pierde. Cuando el jugador está satisfecho con su conjunto actual de cartas, "se planta" (es decir, deja de tomar cartas) y se revela la carta oculta del repartidor. Si el total del repartidor es 16 o menos, debe tomar otra carta; en caso contrario, el repartidor debe plantarse. El repartidor debe seguir tomando cartas hasta que la suma de todas sus cartas sea mayor o igual a 17. Si el repartidor se pasa de 21, el jugador gana. En caso contrario, la mano con el total de puntos que sea mayor gana. Si el repartidor y el jugador tienen el mismo total de puntos, el juego es un "empate" y nadie gana. Observe que el valor de un as para un repartidor depende de la(s) otra(s) carta(s) del repartidor y de las reglas de la casa del casino. Por lo general, un repartidor debe obtener totales de 16 o menos y debe "plantarse" al obtener totales de 17 o más. Sin embargo, para un "suave 17" (una mano con un total de 17, en donde un as cuenta como 11), algunos casinos requieren que el repartidor tome otra carta y otros requieren que se plante (nosotros vamos a requerir que el repartidor se plante). A dicha mano se le conoce como "suave 17", ya que al tomar otra carta la mano no puede "pasarse".*

El servicio Web (figura 28.13) almacena cada carta como un objeto `String` que consiste en un número del `1` al `13`, para representar la cara de la carta (del as al rey, respectivamente), seguido de un espacio y un dígito del `0` al `3`, que representa el palo de la carta (corazones, diamantes, bastos o espadas, respectivamente). Por ejemplo, el joto de bastos se representa como `"11 2"` y el dos de corazones se representa como `"2 0"`. Para crear y desplegar este servicio Web, siga los pasos que se presentan en las secciones 28.3.3 a 28.3.4 para el servicio `EnteroEnorme`.

```java
// Fig. 28.13: Blackjack.java
// Servicio Web Blackjack que reparte cartas y evalúa manos
package com.deitel.jhtp7.ch28.blackjack;

import java.util.ArrayList;
import java.util.Random;
import javax.annotation.Resource;
import javax.jws.WebService;
import javax.jws.WebMethod;
import javax.jws.WebParam;
import javax.servlet.http.HttpSession;
import javax.servlet.http.HttpServletRequest;
import javax.xml.ws.WebServiceContext;
import javax.xml.ws.handler.MessageContext;

@WebService( name = "Blackjack", serviceName = "ServicioBlackjack" )
public class Blackjack
```

Figura 28.13 | Servicio Web `Blackjack` que reparte cartas y evalúa manos. (Parte 1 de 3).

```
{
   // usa @Resource para crear un objeto WebServiceContext para rastrear sesiones
   private @Resource WebServiceContext contextoServicioWeb;
   private MessageContext contextoMensaje; // se utiliza en el rastreo de sesiones
   private HttpSession sesion; // almacena los atributos de la sesión

   // reparte una carta
   @WebMethod( operationName = "repartirCarta" )
   public String repartirCarta()
   {
      String carta = "";

      ArrayList< String > mazo =
         ( ArrayList< String > ) sesion.getAttribute( "mazo" );

      carta = mazo.get( 0 ); // obtiene la carta superior del mazo
      mazo.remove( 0 ); // elimina la carta superior del mazo

      return carta;
   } // fin del Método Web repartirCarta

   // baraja el mazo
   @WebMethod( operationName = "barajar" )
   public void barajar()
   {
      // obtiene el objeto HttpSession para almacenar el mazo para el cliente actual
      contextoMensaje = contextoServicioWeb.getMessageContext();
      sesion = ( ( HttpServletRequest ) contextoMensaje.get(
         MessageContext.SERVLET_REQUEST ) ).getSession();

      // llena el mazo de cartas
      ArrayList< String > mazo = new ArrayList< String >();

      for ( int cara = 1; cara <= 13; cara++ ) // itera a través de las caras
         for ( int palo = 0; palo <= 3; palo++ ) // itera a través de los palos
            mazo.add( cara + " " + palo ); // agrega cada carta al mazo

      String cartaTemp; // guarda la carta temporalmente durante el intercambio
      Random objetoAleatorio = new Random(); // genera números aleatorios
      int indice; // índice la carta seleccionada al azar

      for ( int i = 0; i < mazo.size() ; i++ ) // baraja
      {
         indice = objetoAleatorio.nextInt( mazo.size() - 1 );

         // intercambia la carta en la posición i con la carta seleccionada al azar
         cartaTemp = mazo.get( i );
         mazo.set( i, mazo.get( indice ) );
         mazo.set( indice, cartaTemp );
      } // fin de for

      // agrega este mazo a la sesión del usuario
      sesion.setAttribute( "mazo", mazo );
   } // fin del Método Web barajar

   // determina el valor de una mano
   @WebMethod( operationName = "obtenerValorMano" )
   public int obtenerValorMano( @WebParam( name = "mano" ) String mano )
   {
```

Figura 28.13 | Servicio Web `Blackjack` que reparte cartas y evalúa manos. (Parte 2 de 3).

```
77          // divide la mano en cartas
78          String[] cartas = mano.split( "\t" );
79          int total = 0; // valor total de las cartas en la mano
80          int cara; // cara de la carta actual
81          int cuentaAses = 0; // número de ases en la mano
82
83          for ( int i = 0; i < cartas.length; i++ )
84          {
85             // analiza la cadena y obtiene el primer int en el objeto String
86             cara = Integer.parseInt(
87                cartas[ i ].substring( 0, cartas[ i ].indexOf( " " ) ) );
88
89             switch ( cara )
90             {
91                case 1: // en as, incrementa cuentaAses
92                   ++cuentaAses;
93                   break;
94                case 11: // joto
95                case 12: // reina
96                case 13: // rey
97                   total += 10;
98                   break;
99                default: // en cualquier otro caso, suma la cara
100                  total += cara;
101                  break;
102            } // fin de switch
103         } // fin de for
104
105         // calcula el uso óptimo de los ases
106         if ( cuentaAses > 0 )
107         {
108            // si es posible, cuenta un as como 11
109            if ( total + 11 + cuentaAses - 1 <= 21 )
110               total += 11 + cuentaAses - 1;
111            else // en cualquier otro caso, cuenta todos los ases como 1
112               total += cuentaAses;
113         } // fin de if
114
115         return total;
116      } // fin del Método Web obtenerValorMano
117   } // fin de la clase Blackjack
```

Figura 28.13 | Servicio Web `Blackjack` que reparte cartas y evalúa manos. (Parte 3 de 3).

Rastreo de sesiones en servicios Web

El cliente del servicio Web `Blackjack` llama primero al método `barajar` (líneas 40 a 71) para barajar el mazo de cartas. Este método también coloca el mazo de cartas en un objeto `HttpSession` que es específico para el cliente que llamó a `barajar`. Para usar el rastreo de sesiones en un servicio Web, debemos incluir código para los recursos que mantienen la información de estado de la sesión. Anteriormente, había que escribir el código, que algunas veces era tedioso, para crear estos recursos. Sin embargo, JAX-WS se encarga de esto por nosotros mediante la **anotación `@Resource`**. Esta anotación permite a herramientas como Netbeans "inyectar" el código complejo de soporte en nuestra clase, con lo cual nos podemos enfocar en la lógica de negocios, en vez de hacerlo en el código de soporte. El concepto de usar anotaciones para agregar código que de soporte a nuestras clases se conoce como **inyección de dependencias**. Las anotaciones como `@WebService`, `@WebMethod` y `@Webparam` también realizan la inyección de dependencias.

En la línea 20 se inyecta un objeto `WebServiceContext` en nuestra clase. Un objeto **`WebServiceContext`** permite a un servicio Web acceder a la información para una petición específica y darle mantenimiento, como el

estado de la sesión. A medida que analice el código de la figura 28.13 observará que nunca creamos el objeto WebServiceContext. Todo el código necesario para crearlo se inyecta en la clase mediante la anotación @Resource. En la línea 21 se declara una variable del tipo de interfaz **MessageContext**, que el servicio Web utilizará para obtener un objeto HttpSession para el cliente actual. En la línea 22 se declara la variable HttpSession que el servicio Web utilizará para manipular la información de estado de la sesión.

En la línea 44 del método barajar se utiliza el objeto conextoServicioWeb que se inyectó en la línea 20 para obtener un objeto MessageContext. Después, en las líneas 45 y 46 se utiliza el método **get** del objeto MessageContext para obtener el objeto HttpSession para el cliente actual. El método **get** recibe una constante que indica lo que se debe obtener del objeto MessageContext. En este caso, la constante MessageContext.SERVLET_REQUEST indica que nos gustaría obtener el objeto HttpServletRequest para el cliente actual. Después llamamos al método getSession para obtener el objeto HttpSession del objeto HttpServletRequest.

En las líneas 49 a 70 se genera un objeto ArrayList que representa a un conjunto de cartas, se baraja el mazo y se almacena en el objeto sesion del cliente. En las líneas 51 a 53 se utilizan ciclos anidados para generar objetos String en la forma "*cara palo*" para representar a cada una de las posibles cartas en el mazo. En las líneas 59 a 67 se baraja el mazo, intercambiando cada carta con otra seleccionada al azar. En la línea 70 se inserta el objeto ArrayList en el objeto sesion para mantener el mazo entre las llamadas a los métodos desde un cliente específico.

En las líneas 25 a 37 se define el método repartirCarta como un método Web. En las líneas 30 y 31 se utiliza el objeto sesion para obtener el atributo de sesión "mazo" que se almacenó en la línea 70 del método barajar. El método getAttribute recibe como parámetro un objeto String que identifica el objeto Object a obtener del estado de la sesión. El objeto HttpSession puede almacenar muchos Object, siempre y cuando cada uno tenga un identificador único. Observe que se debe llamar al método barajar antes de llamar al método repartirCarta por primera vez para un cliente; en caso contrario, se producirá una excepción en las líneas 30 y 31, ya que getAttribute devolverá null. Después de obtener el mazo del usuario, repartirCarta obtiene la carta superior del mazo (línea 33), la quita del mazo (línea 34) y devuelve el valor de la carta como un objeto String (línea 36). Sin usar el rastreo de sesiones, el mazo de cartas tendría que pasarse entre cada una de las llamadas a los métodos. El rastreo de sesiones facilita el proceso de llamar al método repartirCarta (no requiere argumentos) y elimina la sobrecarga de enviar el mazo a través de la red varias veces.

El método obtenerValorMano (líneas 74 a 116) determina el valor total de las cartas en una mano, al tratar de obtener la mayor puntuación posible sin pasar de 21. Recuerde que un as se puede contar como 1 u 11, y todas las cartas con cara cuentan como 10. Este método no utiliza el objeto sesion, ya que el mazo de cartas no se utiliza en este método.

Como veremos pronto, la aplicación cliente mantiene una mano de cartas como un objeto String, en el cual cada carta va separada de un carácter de tabulación. En la línea 78 se divide en tokens la mano de cartas (representada por repartir) en cartas individuales, mediante una llamada al método split de String, y se pasa a un objeto String que contiene los caracteres delimitadores (en este caso, sólo un tabulador). El método Split usa los caracteres delimitadores para separar los tokens en el objeto String. En las líneas 83 a 103 se cuenta el valor de cada carta. En las líneas 86 y 87 se obtiene el primer entero (la cara) y se utiliza ese valor en la instrucción switch (líneas 89 a 102). Si la carta es un as, el método incrementa la variable cuentaAses. En breve hablaremos acerca de cómo se utiliza esta variable. Si la carta es un 11, 12 o 13 (joto, reina o rey), el método suma 10 al valor total de la mano (línea 97). Si la carta es cualquier otra cosa, el método incrementa el total en base a ese valor (línea 100).

Como un as puede tener uno de dos valores, se requiere lógica adicional para procesar los ases. En las líneas 106 a 113 del método obtenerValorMano se procesan los ases después de todas las demás cartas. Si una mano contiene varios ases, sólo un as puede contarse como 11. La condición en la línea 109 determina si el contar un as como 11 y el resto como 1 producirá un total que no exceda a 21. Si esto es posible, en la línea 110 se ajusta el total de manera acorde. En caso contario, en la línea 112 se ajusta el total, y se cuenta cada as como 1.

El método obtenerValorMano incrementa al máximo el valor de las cartas actuales sin exceder a 21. Por ejemplo, imagine que el repartidor tiene un 7 y recibe un as. El nuevo total podría ser 8 o 18. Sin embargo, obtenerValorMano siempre incrementa al máximo el valor de las cartas sin pasar de 21, por lo que el nuevo total es 18.

28.6.2 Cómo consumir el servicio Web Blackjack

Ahora usaremos el servicio Web Blackjack en una aplicación de java (figura 28.14). La aplicación lleva la cuenta de las cartas del jugador y del repartidor, y el servicio Web lleva la cuenta de las cartas que se han repartido.

El constructor (líneas 34 a 83) establece la GUI (línea 36), modifica el color de fondo de la ventana (línea 40) y crea el objeto proxy del servicio Web `Blackjack` (líneas 46 y 47). En la GUI, cada jugador tiene 11 objetos `JLabel`: el máximo número de cartas que se pueden repartir sin exceder automáticamente a 21 (es decir, cuatro ases, cuatro de dos y tres de tres). Estos objetos `JLabel` se colocan en un objeto `ArrayList` de objetos `JLabel` (líneas 59 a 82), para que podamos indizar el objeto `ArrayList` durante el juego y determinar el objeto `JLabel` que mostrará una imagen de una carta específica.

En el marco de trabajo JAX-WS 2.0, el cliente debe indicar si desea permitir al servicio Web mantener la información de la sesión. En las líneas 50 y 51 del constructor se lleva a cabo esta tarea. Primero convertimos el objeto proxy al tipo de interfaz `BindingProvider`. Un objeto **`BindingProvider`** permite al cliente manipular la información de petición que se enviará al servidor. Esta información se almacena en un objeto que implementa a la interfaz **`RequestContext`**. Los objetos `BindingProvider` y `RequestContext` son parte del marco de trabajo que crea el IDE cuando agregamos un cliente de servicio Web a la aplicación. A continuación, en las líneas 50 y 51 se invoca el **método `getRequestContext`** de `BindingProvider` para obtener el objeto `RequestContext`. Luego se hace una llamada al **método `put`** de `RequestContext` para establecer la propiedad `BindingProvider.SESSION_MAINTAIN_PROPERTY` a `true`, lo cual permite el rastreo de sesiones desde el lado cliente, de manera que el servicio Web sepa qué cliente está invocando a sus métodos.

```
1  // Fig. 28.14: JuegoBlackjackJFrame.java
2  // Juego de Blackjack que utiliza el servicio Web Blackjack
3  package com.deitel.jhtp7.ch28.clienteblackjack;
4
5  import java.awt.Color;
6  import java.util.ArrayList;
7  import javax.swing.ImageIcon;
8  import javax.swing.JLabel;
9  import javax.swing.JOptionPane;
10 import javax.xml.ws.BindingProvider;
11 import com.deitel.jhtp7.cap28.clienteblackjack.Blackjack;
12 import com.deitel.jhtp7.cap28.clienteblackjack.ServicioBlackjack;
13
14 public class JuegoBlackjackJFrame extends javax.swing.JFrame
15 {
16    private String cartasJugador;
17    private String cartasRepartidor;
18    private ArrayList< JLabel > naipes; // lista de objetos JLabel con imágenes de las
      cartas
19    private int cartaActualJugador; // número de carta actual del jugador
20    private int cartaActualRepartidor; // número de carta actual de proxyBlackjack
21    private ServicioBlackjack servicioBlackjack; // se utiliza para obtener el proxy
22    private Blackjack proxyBlackjack; // se utiliza para acceder al servicio Web
23
24    // enumeración de estados del juego
25    private enum EstadoJuego
26    {
27       EMPATE, // el juego termina en un empate
28       PIERDE, // el jugador pierde
29       GANA, // el jugador gana
30       BLACKJACK // el jugador tiene blackjack
31    } // fin de enum EstadoJuego
32
33    // constructor sin argumentos
34    public JuegoBlackjackJFrame()
35    {
36       initComponents();
```

Figura 28.14 | Juego de Blackjack que utiliza el servicio Web `Blackjack`. (Parte 1 de 9).

```
      // debido a un error en Netbeans, debemos cambiar el color de fondo
      // del objeto JFrame aquí, en vez de hacerlo en el diseñador
      getContentPane().setBackground( new Color( 0, 180, 0 ) );

      // inicializa el proxy blackjack
      try
      {
         // crea los objetos para acceder al servicio Web Blackjack
         servicioBlackjack = new ServicioBlackjack();
         proxyBlackjack = servicioBlackjack.getBlackjackPort();

         // habilita el rastreo de sesiones
         ( ( BindingProvider ) proxyBlackjack ).getRequestContext().put(
            BindingProvider.SESSION_MAINTAIN_PROPERTY, true );
      } // fin de try
      catch ( Exception e )
      {
         e.printStackTrace();
      } // fin de catch

      // agrega componentes JLabel al objeto ArrayList naipes para manipularlo mediante
      programación
      naipes = new ArrayList< JLabel >();

      naipes.add( 0, carta1RepartidorJLabel );
      naipes.add( carta2RepartidorJLabel );
      naipes.add( carta3RepartidorJLabel );
      naipes.add( carta4RepartidorJLabel );
      naipes.add( carta5RepartidorJLabel );
      naipes.add( carta6RepartidorJLabel );
      naipes.add( carta7RepartidorJLabel );
      naipes.add( carta8RepartidorJLabel );
      naipes.add( carta9RepartidorJLabel );
      naipes.add( carta10RepartidorJLabel );
      naipes.add( carta11RepartidorJLabel );
      naipes.add( carta1JugadorJLabel );
      naipes.add( carta2JugadorJLabel );
      naipes.add( carta3JugadorJLabel );
      naipes.add( carta4JugadorJLabel );
      naipes.add( carta5JugadorJLabel );
      naipes.add( carta6JugadorJLabel );
      naipes.add( carta7JugadorJLabel );
      naipes.add( carta8JugadorJLabel );
      naipes.add( carta9JugadorJLabel );
      naipes.add( carta10JugadorJLabel );
      naipes.add( carta11JugadorJLabel );
   } // fin del constructor sin argumentos

   // juega la mano del repartidor
   private void juegoRepartidor()
   {
      try
      {
         // mientras el valor de la mano del repartidor sea menor a 17
         // el repartidor debe seguir tomando cartas
         String[] cartas = cartasRepartidor.split( "\t" );

         // muestra las cartas del repartidor
```

Figura 28.14 | Juego de Blackjack que utiliza el servicio Web `Blackjack`. (Parte 2 de 9).

```
         for ( int i = 0; i < cartas.length; i++ )
            mostrarCarta( i, cartas[ i ] );

         while ( proxyBlackjack.obtenerValorMano( cartasRepartidor ) < 17 )
         {
            String nuevaCarta = proxyBlackjack.repartirCarta(); // reparte una nueva carta
            cartasRepartidor += "\t" + nuevaCarta; // reparte una nueva carta
            mostrarCarta( cartaActualRepartidor, nuevaCarta );
            ++cartaActualRepartidor;
            JOptionPane.showMessageDialog( this, "El repartidor toma una carta",
               "Turno del repartidor", JOptionPane.PLAIN_MESSAGE );
         } // end while

         int totalRepartidor = proxyBlackjack.obtenerValorMano( cartasRepartidor );
         int totalJugador = proxyBlackjack.obtenerValorMano( cartasJugador );

         // si el repartidor se pasó, el jugador gana
         if ( totalRepartidor > 21 )
         {
            finDelJuego( EstadoJuego.GANA );
            return;
         } // fin de if

         // si el repartidor y el jugador tienen menos de 21
         // la mayor puntuación gana, si tienen igual puntuación es un empate
         if ( totalRepartidor > totalJugador )
            finDelJuego( EstadoJuego.PIERDE );
         else if (totalRepartidor < totalJugador )
            finDelJuego( EstadoJuego.GANA );
         else
            finDelJuego( EstadoJuego.EMPATE );
      } // fin de try
      catch ( Exception e )
      {
         e.printStackTrace();
      } // fin de catch
   } // fin del método juegoRepartidor

   // muestra la carta representada por valorCarta en el objeto JLabel especificado
   public void mostrarCarta( int carta, String valorCarta )
   {
      try
      {
         // obtiene el objeto JLabel correcto de naipes
         JLabel mostrarEtiqueta = naipes.get( carta );

         // si la cadena que representa la carta está vacía, muestra la parte posterior de la carta
         if ( valorCarta.equals( "" ) )
         {
            mostrarEtiqueta.setIcon( new ImageIcon( getClass().getResource(
               "/com/deitel/jhtp7/cap28/clienteblackjack/" +
               "blackjack_imagenes/cartpost.png" ) ) ) ;
            return;
         } // fin de if

         // obtiene el valor de la cara de la carta
         String cara = valorCarta.substring( 0, valorCarta.indexOf( " " ) );
```

Figura 28.14 | Juego de Blackjack que utiliza el servicio Web `Blackjack`. (Parte 3 de 9).

```

         // obtiene el palo de la carta
         String palo =
            valorCarta.substring( valorCarta. indexOf( " " ) + 1 );

         char letraPalo; // letra del palo que se usa para formar el archivo de imagen

         switch ( Integer.parseInt( palo ) )
         {
            case 0: // corazones
               letraPalo = 'c';
               break;
            case 1: // diamantes
               letraPalo = 'd';
               break;
            case 2: // bastos
               letraPalo = 'b';
               break;
            default: // espadas
               letraPalo = 'e';
               break;
         } // fin de switch

         // establece la imagen para mostrarEtiqueta
         mostrarEtiqueta.setIcon( new ImageIcon( getClass().getResource(
            "/com/deitel/jhtp7/cap28/clienteblackjack/blackjack_imagenes/" +
            cara + letraPalo + ".png" ) ) );
      } // fin de try
      catch ( Exception e )
      {
         e.printStackTrace();
      } // fin de catch
   } // fin del método mostrarCarta

   // muestra todas las cartas del jugador y el mensaje apropiado
   public void finDelJuego( EstadoJuego ganador )
   {
      String[] cartas = cartasRepartidor.split( "\t" );

      // muestra las cartas de proxyBlackjack
      for ( int i = 0; i < cartas.length; i++ )
         mostrarCarta( i, cartas[ i ]);

      // muestra la imagen del estado apropiado
      if ( ganador == EstadoJuego.GANA )
         estadoJLabel.setText( "Usted gana!" );
      else if ( ganador == EstadoJuego.PIERDE )
         estadoJLabel.setText( "Usted pierde." );
      else if ( ganador == EstadoJuego.EMPATE )
         estadoJLabel.setText( "Es un empate." );
      else // blackjack
         estadoJLabel.setText( "Blackjack!" );

      // muestra las puntuaciones finales
      int totalRepartidor = proxyBlackjack.obtenerValorMano( cartasRepartidor );
      int totalJugador = proxyBlackjack.obtenerValorMano( cartasJugador );
      totalRepartidorJLabel.setText( "Repartidor: " + totalRepartidor );
      totalJugadorJLabel.setText( "Jugador: " + totalJugador );

```

Figura 28.14 | Juego de Blackjack que utiliza el servicio Web `Blackjack`. (Parte 4 de 9).

```
         // restablece para nuevo juego
         pasarJButton.setEnabled( false );
         pedirJButton.setEnabled( false );
         repartirJButton.setEnabled( true );
      } // fin del método finDelJuego

      // El método initComponents es generado automáticamente por Netbeans y se llama
      // desde el constructor para inicializar la GUI. No mostraremos aquí este método
      // para ahorrar espacio. Abra JuegoBlackjackJFrame.java en la carpeta de
      // este ejemplo para ver el código generado completo (líneas 221 a 531)

      // maneja el clic del objeto pasarJButton
      private void pasarJButtonActionPerformed(
         java.awt.event.ActionEvent evt )
      {
         pasarJButton.setEnabled( false );
         pedirJButton.setEnabled( false );
         repartirJButton.setEnabled( true );
         juegoRepartidor();
      } // fin del método pasarJButtonActionPerformed

      // maneja el clic del objeto pedirJButton
      private void pedirJButtonActionPerformed(
         java.awt.event.ActionEvent evt )
      {
         // obtiene otra carta para el jugador
         String carta = proxyBlackjack.repartirCarta(); // reparte una nueva carta
         cartasJugador += "\t" + carta; // agrega la carta a la mano

         // actualiza la GUI para mostrar una nueva carta
         mostrarCarta( cartaActualJugador, carta );
         ++cartaActualJugador;

         // determina el nuevo valor de la mano del jugador
         int total = proxyBlackjack.obtenerValorMano( cartasJugador );

            if ( total > 21 ) // el jugador se pasa
               finDelJuego( EstadoJuego.PIERDE );
            if ( total == 21 ) // el jugador no puede tomar más cartas
          {
             pedirJButton.setEnabled( false );
             juegoRepartidor();
          } // fin de if
       } // fin del método pedirJButtonActionPerformed

      // maneja el clic del objeto repartirJButton
      private void repartirJButtonActionPerformed(
         java.awt.event.ActionEvent evt )
      {
         String carta; // almacena una carta temporalmente hasta que se agrega a una mano

         // borra las imágenes de las cartas
         for ( int i = 0; i < naipes.size(); i++ )
            naipes.get( i ).setIcon( null );

         estadoJLabel.setText( "" );
         totalRepartidorJLabel.setText( "" );
         totalJugadorJLabel.setText( "" );

```

Figura 28.14 | Juego de Blackjack que utiliza el servicio Web `Blackjack`. (Parte 5 de 9).

```
      // crea un nuevo mazo barajado en un equipo remoto
      proxyBlackjack.barajar();

      // reparte dos cartas al jugador
      cartasJugador = proxyBlackjack.repartirCarta(); // agrega la primera carta a la mano
      mostrarCarta( 11, cartasJugador ); // muestra la primera carta
      carta = proxyBlackjack.repartirCarta(); // reparte la segunda carta
      mostrarCarta( 12, carta ); // muestra la segunda carta
      cartasJugador += "\t" + carta; // agrega la segunda carta a la mano

      // reparte dos cartas a proxyBlackjack, pero sólo muestra la primera
      cartasRepartidor = proxyBlackjack.repartirCarta(); // agrega la primera carta a la mano
      mostrarCarta( 0, cartasRepartidor ); // muestra la primera carta
      carta = proxyBlackjack.repartirCarta(); // reparte la segunda carta
      mostrarCarta( 1, "" ); // muestra la parte posterior de la carta
      cartasRepartidor += "\t" + carta; // agrega la segunda carta a la mano

      pasarJButton.setEnabled( true );
      pedirJButton.setEnabled( true );
      repartirJButton.setEnabled( false );

      // determina el valor de las dos manos
      int totalRepartidor = proxyBlackjack.obtenerValorMano( cartasRepartidor );
      int totalJugador = proxyBlackjack.obtenerValorMano( cartasJugador );

      // si ambas manos son iguales a 21, es un empate
      if ( totalJugador == totalRepartidor && totalJugador == 21 )
         finDelJuego( EstadoJuego.EMPATE );
      else if (totalRepartidor == 21 ) // proxyBlackjack tiene blackjack
         finDelJuego( EstadoJuego.PIERDE );
      else if (totalJugador == 21 ) // blackjack
         finDelJuego( EstadoJuego.BLACKJACK );

      // la siguiente carta para proxyBlackjack tiene el índice 2
      cartaActualRepartidor = 2;

      // la siguente carta para el jugador tiene el índice 13
      cartaActualJugador = 13;
   } // fin del método repartirJButtonActionPerformed

   // empieza la ejecución de la aplicación
   public static void main( String args[] )
   {
      java.awt.EventQueue.invokeLater(
         new Runnable()
         {
            public void run()
            {
               new JuegoBlackjackJFrame().setVisible(true);
            }
         }
      ); // fin de la llamada a java.awt.EventQueue.invokeLater
   } // fin del método main

   // Declaración de variables - no modificar
   private javax.swing.JLabel carta10JugadorJLabel;
   private javax.swing.JLabel carta10RepartidorJLabel;
   private javax.swing.JLabel carta11JugadorJLabel;
```

Figura 28.14 | Juego de Blackjack que utiliza el servicio Web `Blackjack`. (Parte 6 de 9).

```
    private javax.swing.JLabel carta11RepartidorJLabel;
    private javax.swing.JLabel carta1JugadorJLabel;
    private javax.swing.JLabel carta1RepartidorJLabel;
    private javax.swing.JLabel carta2JugadorJLabel;
    private javax.swing.JLabel carta2RepartidorJLabel;
    private javax.swing.JLabel carta3JugadorJLabel;
    private javax.swing.JLabel carta3RepartidorJLabel;
    private javax.swing.JLabel carta4JugadorJLabel;
    private javax.swing.JLabel carta4RepartidorJLabel;
    private javax.swing.JLabel carta5JugadorJLabel;
    private javax.swing.JLabel carta5RepartidorJLabel;
    private javax.swing.JLabel carta6JugadorJLabel;
    private javax.swing.JLabel carta6RepartidorJLabel;
    private javax.swing.JLabel carta7JugadorJLabel;
    private javax.swing.JLabel carta7RepartidorJLabel;
    private javax.swing.JLabel carta8JugadorJLabel;
    private javax.swing.JLabel carta8RepartidorJLabel;
    private javax.swing.JLabel carta9JugadorJLabel;
    private javax.swing.JLabel carta9RepartidorJLabel;
    private javax.swing.JLabel estadoJLabel;
    private javax.swing.JLabel jugadorJLabel;
    private javax.swing.JButton pasarJButton;
    private javax.swing.JButton pedirJButton;
    private javax.swing.JLabel repartidorJLabel;
    private javax.swing.JButton repartirJButton;
    private javax.swing.JLabel totalJugadorJLabel;
    private javax.swing.JLabel totalRepartidorJLabel;
    // End of variables declaration
} // fin de la clase JuegoBlackjackJFrame
```

a) Manos del repartidor y del jugador, después de que el usuario hace clic en el botón **Repartir**.



Figura 28.14 | Juego de Blackjack que utiliza el servicio Web `Blackjack`. (Parte 7 de 9).

b) Manos del repartidor y del jugador, después de que el usuario hace clic en **Pedir** dos veces y luego en **Pasar**. En este caso, el jugador gana.

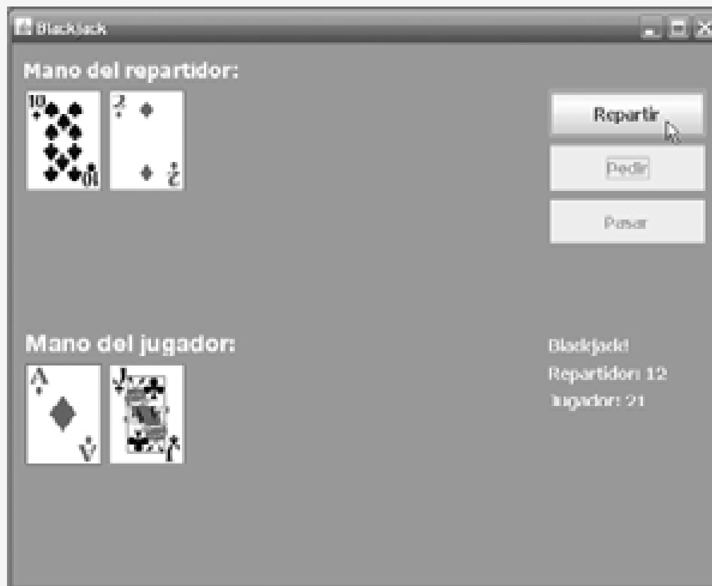


c) Manos del repartidor y del jugador, después de que el usuario hace clic en el botón **Pasar** con base en la mano inicial. En este caso, el jugador pierde.



Figura 28.14 | Juego de Blackjack que utiliza el servicio Web `Blackjack`. (Parte 8 de 9).

d) Manos del repartidor y del jugador, después de que al usuario se le reparte blackjack.



e) Manos del repartidor y del jugador, después de que al repartidor se le reparte blackjack.



Figura 28.14 | Juego de Blackjack que utiliza el servicio Web `Blackjack`. (Parte 9 de 9).

El método finDelJuego (líneas 178 a 215) muestra todas las cartas del repartidor, muestra el mensaje apropiado en estadoJLabel y muestra los totales finales en puntos del repartidor y del jugador. El método finDelJuego recibe como argumento un miembro de la enumeración EstadoJuego (definida en las líneas 25 a 31). La enumeración representa si el jugador empató, perdió o ganó el juego; sus cuatro miembros son EMPATE, PIERDE, GANA y BLACKJACK.

Cuando el jugador hace clic en el botón **Repartir**, el método repartirJButtonActionPerformed (líneas 567 a 618) borra todos los objetos JLabel que muestran cartas o información sobre el estado del juego. A continuación, el mazo se baraja (línea 581), y el jugador y el repartidor reciben dos cartas cada uno (líneas 584 a 595). Después, en las líneas 602 y 603 se calcula el total de cada mano. Si el jugador y el repartidor obtienen puntuaciones de 21, el programa llama al método finDelJuego, y le pasa EstadoJuego.EMPATE (línea 607). Si sólo el repartidor tiene 21, el programa pasa EstadoJuego.PIERDE al método finDelJuego (línea 609). Si sólo el jugador tiene 21 después de repartir las primeras dos cartas, el programa pasa EstadoJuego.BLACKJACK al método finDelJuego (línea 611).

Si repartirJButtonActionPerformed no llama a finDelJuego, el jugador puede tomar más cartas haciendo clic en el botón **Pedir**, el cual llama a pedirJButtonActionPerformed en las líneas 543 a 564. Cada vez que un jugador hace clic en **Pedir**, el programa reparte una carta más al jugador y la muestra en la GUI. Si el jugador excede a 21, el juego termina y el jugador pierde. Si el jugador tiene exactamente 21, no puede tomar más cartas y se hace una llamada al método juegoRepartidor (líneas 86 a 131), lo cual provoca que el repartidor tome cartas hasta que su mano tenga un valor de 17 o más (líneas 98 a 106). Si el repartidor excede a 21, el jugador gana (línea 114); en caso contrario, se comparan los valores de las manos, y se hace una llamada a finDelJuego con el argumento apropiado (líneas 120 a 125).

Al hacer clic en el botón **Pasar**, estamos indicando que el jugador no desea recibir otra carta. El método pasarJButtonActionPerformed (líneas 533 a 540) deshabilita los botones **Pedir** y **Pasar**, habilita el botón **Repartir** y después llama al método juegoRepartidor.

El método mostrarCarta (líneas 134 a 184) actualiza la GUI para mostrar una carta recién repartida. El método recibe como argumentos un índice entero para el objeto JLabel en el objeto ArrayList que debe tener establecida su imagen, y un objeto String que representa a la carta. Un objeto String vacío indica que deseamos mostrar la cara de la carta hacia abajo. Si el método mostrarCarta recibe un objeto String que no esté vacío, el programa extrae la cara y el palo del objeto String, y utiliza esta información para mostrar la imagen correcta. La instrucción switch (líneas 159 a 173) convierte el número que representa el palo en un entero, y asigna el carácter apropiado a cartaPalo (c para corazones, d para diamantes, b para bastos y e para espadas). El carácter en letraPalo se utiliza para completar el nombre del archivo de imagen (líneas 176 a 178).

En este ejemplo, aprendió a configurar un servicio Web para dar soporte al manejo de sesiones, de manera que pueda llevar la cuenta del estado de la sesión de cada cliente. También aprendió a indicar desde una aplicación de escritorio cliente que desea participar en el rastreo de sesiones. Ahora aprenderá a acceder a una base de datos desde un servicio Web, y cómo consumir un servicio Web desde una aplicación Web cliente.

28.7 Cómo consumir un servicio Web controlado por base de datos desde una aplicación Web

Nuestros ejemplos anteriores acceden a los servicios Web desde aplicaciones de escritorio creadas en Netbeans. Sin embargo, podemos utilizarlos con igual facilidad en aplicaciones Web creadas con Netbeans o Sun Java Studio Creator 2. De hecho, y como los comercios basados en Internet prevalecen cada vez más, es común que las aplicaciones Web consuman servicios Web. En esta sección le presentaremos un servicio Web de reservación de una aerolínea, que recibe información sobre el tipo de asiento que desea reservar un cliente, y hace una reservación si está disponible dicho asiento. Más adelante en esta sección, le presentaremos una aplicación Web que permite a un cliente especificar una petición de reservación, y después utiliza el servicio Web de reservación de una aerolínea para tratar de ejecutar la petición. Utilizaremos Sun Java Studio Creator 2 para crear la aplicación Web.

28.7.1 Configuración de Java DB en Netbeans y creación de la base de datos Reservacion

En este ejemplo, nuestro servicio Web utiliza una base de datos llamada Reservacion, la cual contiene una sola tabla llamada Asientos para localizar un asiento que coincida con la petición de un cliente. Usted creará la base

de datos `Reservacion` mediante el uso de las herramientas que se proporcionan en Netbeans para crear y manipular bases de datos Java DB.

Cómo agregar una base de datos Java DB

Para agregar un servidor de bases de datos Java DB en Netbeans, realice los siguientes pasos:

1. Seleccione **Tools > Options…** para que aparezca el cuadro de diálogo **Options** de Netbeans.
2. Haga clic en el botón **Advanced Options** para mostrar el cuadro de diálogo **Advanced Options**.
3. En **IDE Configuration**, expanda el nodo **Server and External Tool Settings** y seleccione **Java DB Database**.
4. Si las propiedades de Java DB no están ya configuradas, establezca la propiedad **Java DB Location** a la ubicación de Java DB en su sistema. JDK 6 incluye una versión de Java DB, la cual se ubica en Windows, en el directorio `C:\Archivos de programa\Java\jdk1.6.0\db`. Sun Java System Application Server también se incluye con Java DB en `C:\Sun\AppServer\javadb`. Además, establezca la propiedad **Database Location** a la ubicación en donde desea que se almacenen las bases de datos Java DB.

Creación de una base de datos Java DB

Ahora que está configurado el software de bases de datos, cree una nueva base de datos de la siguiente manera:

1. Seleccione **Tools > Java DB Database > Create Java DB Database…**
2. Escriba el nombre de la base de datos a crear (`Reservacion`), un nombre de usuario (`jhtp7`) y una contraseña (`jhtp7`), y después haga clic en **OK** para crear la base de datos.

Cómo agregar una tabla y datos a la base de datos Asientos

Puede usar la ficha **Runtime** de Netbeans (a la derecha de las fichas **Projects** y **Files**) para crear tablas y ejecutar instrucciones SQL que llenen la base de datos con datos:

1. Haga clic en la ficha **Runtime** de Netbeans y expanda el nodo **Databases**.
2. Netbeans debe estar conectado a la base de datos para ejecutar instrucciones SQL. Si Netbeans ya está conectado, proceda al *paso 3*. Si Netbeans no está conectado a la base de datos, aparecerá el ícono enseguida del URL de la base de datos (`jdbc:derby://localhost:1527/Reservacion`). En este caso, haga clic con el botón derecho del ratón en el ícono y seleccione **Connect…**. Una vez conectado, el ícono cambiará a .
3. Expanda el nodo para la base de datos `Reservacion`, haga clic con el botón derecho del ratón en el nodo **Tables** y seleccione **Create Table…** para que aparezca el cuadro de diálogo **Create Table**. Agregue una tabla llamada `Asientos` a la base de datos, y establezca las columnas `Numero`, `Ubicacion`, `Clase` y `Reservado`, como se muestra en la figura 28.15. Use el botón **Add column** para agregar una fila en el cuadro de diálogo por cada columna en la base de datos.

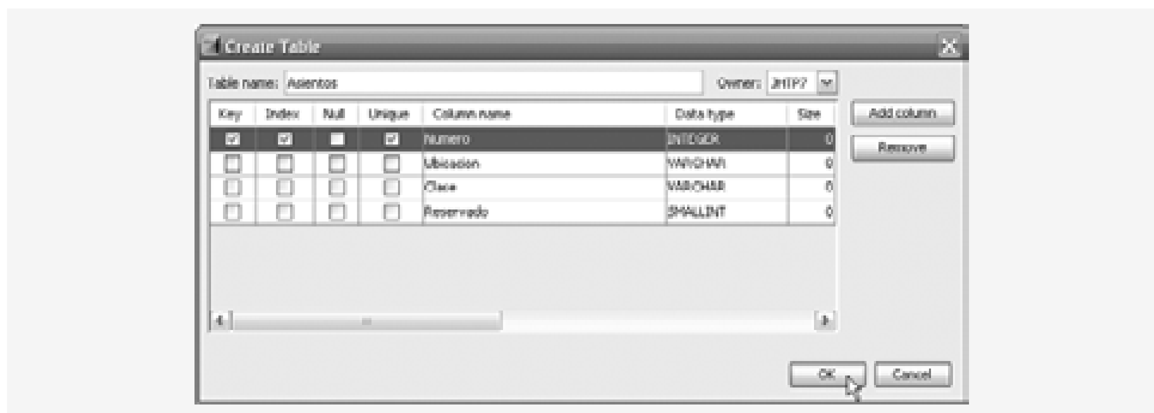


Figura 28.15 | Tabla de configuración de la base de datos `Asientos`.

Número	Ubicación	Clase	Reservado
1	Pasillo	Economica	0
2	Pasillo	Economica	0
3	Pasillo	Primera	0
4	Centro	Economica	0
5	Centro	Economica	0
6	Centro	Primera	0
7	Ventana	Economica	0
8	Ventana	Economica	0
9	Ventana	Primera	0
10	Ventana	Primera	0

Figura 28.16 | Datos de la tabla `Asientos`.

4. A continuación, use comandos `INSERT INTO` para llenar la base de datos con los datos que se muestran en la figura 28.16. Para ello, haga clic con el botón derecho del ratón en la tabla `Asientos` en la ficha **Runtime** y seleccione **Execute Command...** para que aparezca una ficha **SQL Command** en el editor de Netbeans. El archivo `InstruccionesSQLParaLaFig28_16.txt` que se proporciona con los ejemplos de este capítulo contiene los 10 comandos `INSERT INTO` que almacenan los datos que se muestran en la figura 28.16. Sólo copie el texto en ese archivo y péguelo en la ficha **SQL Command**, y después oprima el botón **Run SQL** ( ) a la derecha de la lista desplegable **Connection** en la ficha **SQL Command** para ejecutar los comandos. Para confirmar que los datos se hayan insertado correctamente, haga clic con el botón derecho del ratón en la tabla `Asientos` en la ficha **Runtime**, y seleccione **View Data...**.

Creación del servicio Web Reservacion

Ahora puede crear un servicio Web que utilice la base de datos `Reservacion` (figura 28.17). El servicio Web de reservación de la aerolínea tiene un solo método Web: `reservar` (líneas 25 a 73), el cual busca en una base de datos `Reservacion` que contenga una sola tabla llamada `Asientos` para localizar un asiento que coincida con la petición del usuario. El método recibe dos argumentos: un objeto `String` que representa el tipo de asiento deseado (es decir, `"Ventana"`, `"Centro"` o `"Pasillo"`) y un objeto `String` que representa el tipo de clase deseado (es decir, `"Economica"` o `"Primera"`). Si encuentra un asiento apropiado, el método `reservar` actualiza la base de datos para hacer la reservación y devuelve `true`; en caso contrario, no se realiza la reservación y el método devuelve `false`. Observe que las instrucciones en las líneas 34 a 37 y en las líneas 43 a 44, que consultan y actualizan la base de datos, usan objetos de los tipos `ResultSet` y `Statement` (que vimos en el capítulo 25).

Nuestra base de datos contiene cuatro columnas: el número de asiento (es decir, `1-10`), el tipo de asiento (es decir, `Ventana`, `Centro` o `Pasillo`), el tipo de clase (`Ecnonomica` o `Primera`), y una columna que contiene `1` (verdadero) o `0` (falso) para indicar si el asiento está reservado o no. En las líneas 34 a 37 se obtienen los números de asiento de cualquier asiento disponible que coincida con el asiento y tipo de clase solicitados. Esta instrucción llena el objeto `conjuntoResultados` con los resultados de la consulta

```
SELECT "Numero"
FROM "Asientos"
WHERE ("Reservado" = 0) AND ("Tipo" = tipo) AND ("Clase" = clase)
```

Los parámetros *tipo* y *clase* en la consulta se sustituyen con los valores de los parámetros `tipoAsiento` y `tipoClase` del método `reservar`. Cuando usamos las herramientas de Netbeans para crear una tabla de una base de datos y sus columnas, las herramientas de Netbeans colocan automáticamente la tabla y los nombres de las columnas entre comillas dobles. Por esta razón, debemos colocar la tabla y los nombres de las columnas entre comillas dobles en las instrucciones SQL que interactúan con la base de datos `Reservacion`.

Si `conjuntoResultados` no está vacío (es decir, que por lo menos haya un asiento disponible que coincida con los criterios seleccionados), la condición en la línea 40 es `true` y el servicio Web reserva el primer número de asiento que coincida. Recuerde que el método `next` de `conjuntoResultados` devuelve `true` si existe una fila que no esté vacía, y posiciona el cursor en esa fila. Para obtener el número de asiento (línea 42), accedemos a la primera columna de `conjuntoResultado` (es decir, `conjuntoResultados.getInt(1)`; la primera columna en la fila). Después, en las líneas 43 y 44 se invoca el método `executeUpdate` de `instruccion` para ejecutar el SQL:

```
UPDATE "Asientos"
SET "Reservado" = 1
WHERE ("Numero" = número)
```

el cual marca el asiento como reservado en la base de datos. El parámetro *número* se sustituye con el valor de `numeroAsiento`. El método `reserve` devuelve `true` (línea 45) para indicar que la reservación se realizó con éxito. Si no hay asientos que coincidan, o si ocurrió una excepción, el método `reserve` devuelve `false` (líneas 48, 53, 58 y 70) para indicar que ningún asiento coincidió con la petición del usuario.

```
1  // Fig. 28.17: Reservacion.java
2  // Servicio Web de reservaciones de una aerolínea.
3  package com.deitel.jhtp7.cap28.reservacion;
4
5  import java.sql.Connection;
6  import java.sql.Statement;
7  import java.sql.DriverManager;
8  import java.sql.ResultSet;
9  import java.sql.SQLException;
10 import javax.jws.WebService;
11 import javax.jws.WebMethod;
12 import javax.jws.WebParam;
13
14 @WebService( name = "Reservacion", serviceName = "ServicioReservacion” )
15 public class Reservacion
16 {
17    private static final String URL_BASEDATOS =
18       "jdbc:derby://localhost:1527/Reservacion";
19    private static final String USUARIO = "jhtp7";
20    private static final String CONTRASENIA = "jhtp7";
21    private Connection conexion;
22    private Statement instruccion;
23
24    // un Método Web que puede reservar un asiento
25    @WebMethod( operationName = "reservar" )
26    public boolean reserve( @WebParam( name = "tipoAsiento" ) String tipoAsiento,
27       @WebParam( name = "tipoClase" ) String tipoClase )
28    {
29       try
30       {
31          conexion = DriverManager.getConnection(
32             URL_BASEDATOS, USUARIO, CONTRASENIA );
33          instruccion = conexion.createStatement();
34          ResultSet conjuntoResultados = instruccion.executeQuery(
35             "SELECT \"Numero\" FROM \"Asientos\"" +
36             "WHERE (\"Reservado\" = 0) AND (\"Ubicacion\" = '" + tipoAsiento +
37             "') AND (\"Clase\" = '" + tipoClase + "')" );
38
39          // si el asiento solicitado está disponible, lo reserva
40          if ( conjuntoResultados.next() )
```

Figura 28.17 | Servicio Web de reservaciones de una aerolínea. (Parte 1 de 2).

```
         {
            int asiento = conjuntoResultados.getInt( 1 );
            instruccion.executeUpdate( "UPDATE \"Asientos\" " +
               "SET \"Reservado\" = 1 WHERE \"Numero\" = " + asiento );
            return true;
         } // fin de if

         return false;
      } // fin de try
      catch ( SQLException e )
      {
         e.printStackTrace();
         return false;
      } // fin de catch
      catch ( Exception e )
      {
         e.printStackTrace();
         return false;
      } // fin de catch
      finally
      {
         try
         {
            instruccion.close();
            conexion.close();
         } // fin de try
         catch ( Exception e )
         {
            e.printStackTrace();
            return false;
         } // fin de catch
      } // fin de finally
   } // fin del Método Web reservar
} // fin de la clase Reservacion
```

Figura 28.17 | Servicio Web de reservaciones de una aerolínea. (Parte 2 de 2).

28.7.2 Creación de una aplicación Web para interactuar con el servicio Web `Reservacion`

En esta sección presentaremos una aplicación Web llamada `ClienteReservacion` para consumir el servicio Web `Reservacion`. La aplicación permite a los usuarios seleccionar asientos con base en la clase (`"Economica"` o `"Primera"`) y la ubicación (`"Pasillo"`, `"Centro"` o `"Ventana"`), y después enviar sus peticiones al servicio Web de reservaciones de la aerolínea. Si la petición de la base de datos no tiene éxito, la aplicación instruye al usuario para que modifique la petición e intente de nuevo. La aplicación que presentamos aquí se creó mediante Sun Java Studio Creator 2, JavaServer Faces (JSF) y las técnicas presentadas en los capítulos 26 y 27.

Cómo agregar una referencia de servicio Web a un proyecto en Sun Java Studio Creator 2

Para agregar un servicio Web a una aplicación Web en Java Studio Creator 2, realice los siguientes pasos:

1. Haga clic en el botón **Agregar servicio Web...** ( ) para que aparezca el cuadro de diálogo **Agregar servicio Web**.
2. Haga clic en el botón **Obtener información sobre el servicio Web**.
3. Haga clic en **Agregar** para cerrar el cuadro de diálogo y agregar el proxy del servicio Web a la aplicación Web. Ahora el servicio Web aparecerá en la ficha **Servidores** de Java Studio Creator 2, bajo el nodo **Servicios Web**.

4. Haga clic con el botón derecho del ratón en el nodo **ServicioReservacion** bajo el nodo **Servicios Web**, y seleccione **Agregar a página** para crear una instancia de la clase proxy del servicio Web que puede usar en la clase `Reservar` que proporciona la lógica de la JSP.

Para los fines de este ejemplo, vamos a suponer que usted ya leyó los capítulos 26 y 27, y por ende sabe cómo crear la GUI de una aplicación Web, crear manejadores de eventos, y agregar propiedades al bean de sesión de una aplicación Web (que vimos en la sección 26.4.4).

Reservar.jsp

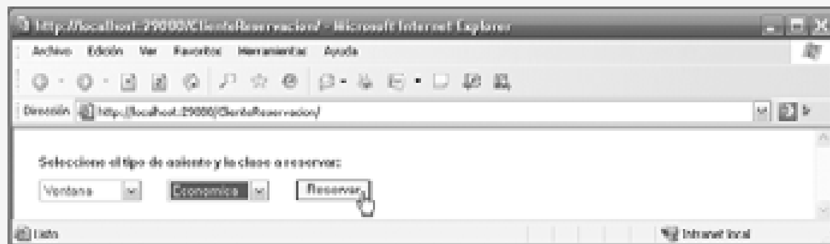
El archivo `Reservar.jsp` (figura 28.18) define dos objetos `ListaDesplegable` y un objeto `Botón`. El objeto `tipoAsientoListaDesplegable` (líneas 26 a 31) muestra todos los tipos de asientos que el usuario puede seleccionar. El objeto `claseListaDesplegable` (líneas 32 a 37) proporciona opciones para el tipo de clase. Los usuarios hacen clic en el objeto `Botón` llamado `reservarBoton` (líneas 38 a 41) para enviar peticiones después de realizar selecciones de los objetos `ListaDesplegable`. Esta página también define tres objetos `Etiqueta: instruccionesEtiqueta` (líneas 21 a 25) para mostrar instrucciones, `exitoEtiqueta` (líneas 42 a 45) para indicar una reservación exitosa, y `errorEtiqueta` (líneas 46 a 50) para mostrar un mensaje apropiado si no hay un asiento disponible que coincida con la selección del usuario. El archivo de bean de página (figura 28.19) adjunta manejadores de eventos a `tipoAsientoListaDesplegable`, `claseListaDesplegable` y `reservarBoton`.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <!-- Fig. 28.18 Reservar.jsp -->
3  <!-- JSP que permite a un usuario seleccionar un asiento -->
4
5  <jsp:root version="1.2" xmlns:f="http://java.sun.com/jsf/core"
6     xmlns:h="http://java.sun.com/jsf/html"
7     xmlns:jsp="http://java.sun.com/JSP/Page"
8     xmlns:ui="http://www.sun.com/web/ui">
9     <jsp:directive.page contentType="text/html;charset=UTF-8"
10       pageEncoding="UTF-8"/>
11    <f:view>
12       <ui:page binding="#{Reservar.page1}" id="page1">
13          <ui:html binding="#{Reservar.html1}" id="html1">
14          <ui:head binding="#{Reservar.head1}" id="head1">
15             <ui:link binding="#{Reservar.link1}" id="link1"
16                url="/resources/stylesheet.css"/>
17          </ui:head>
18          <ui:body binding="#{Reservar.body1}" id="body1"
19             style="-rave-layout: grid">
20             <ui:form binding="#{Reservar.form1}" id="form1">
21                <ui:label binding="#{Reservar.instruccionesEtiqueta}"
22                   id="instruccionesEtiqueta"
23                   style="position: absolute; left: 24px; top: 24px"
24                   text="Seleccione el tipo de asiento y la clase a
25                   reservar:"/>
26                <ui:dropDown binding="#{Reservar.tipoAsientoListaDesplegable}"
27                   id="tipoAsientoListaDesplegable" items=
28                   "#{Reservar.tipoAsientoListaDesplegableDefaultOptions.options}"
29                   style="left: 24px; top: 48px; position: absolute;
30                   width: 96px" valueChangeListener=
31                   "#{Reservar.tipoAsientoListaDesplegable_processValueChange}"/>
32                <ui:dropDown binding="#{Reservar.claseListaDesplegable}"
33                   id="claseListaDesplegable" items=
34                   "#{Reservar.claseListaDesplegableDefaultOptions.options}"
35                   style="left: 144px; top: 48px; position: absolute;
36                   width: 96px" valueChangeListener=
37                   "#{Reservar.claseListaDesplegable_processValueChange}"/>
```

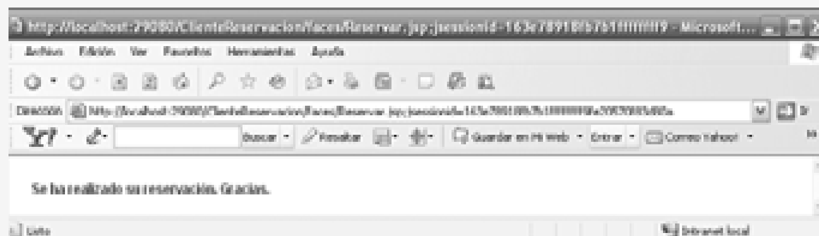
Figura 28.18 | JSP que permite a un usuario seleccionar un asiento. (Parte 1 de 3).

```
               <ui:button action="#{Reservar.reservarBoton_action}"
                  binding="#{Reservar.reservarBoton}" id="reservarBoton"
                  primary="true" style="position: absolute; left: 263px;
                  top: 48px" text="Reservar"/>
               <ui:label binding="#{Reservar.exitoEtiqueta}"
                  id="exitoEtiqueta" style="position: absolute; left: 24px;
                  top: 24px" text="Se ha realizado su reservación.
                  Gracias." visible="false"/>
               <ui:label binding="#{Reservar.errorEtiqueta}" id="errorEtiqueta"
                  style="color: red; left: 24px; top: 96px;
                  position: absolute" text="Este tipo de asiento no está
                  disponible. Modifique su petición e intente de nuevo."
                  visible="false"/>
            </ui:form>
         </ui:body>
         </ui:html>
      </ui:page>
   </f:view>
</jsp:root>
```

a) Selección de un asiento:



b) El asiento se reservó con éxito



c) Ningún asiento coincide con el tipo y clase solicitados:

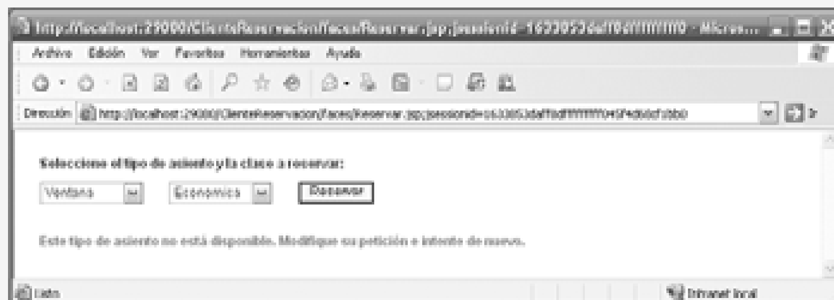


Figura 28.18 | JSP que permite a un usuario seleccionar un asiento. (Parte 2 de 3).

d) Intento de reservar otro asiento de ventana en clase económica, cuando no hay dichos asientos disponibles:

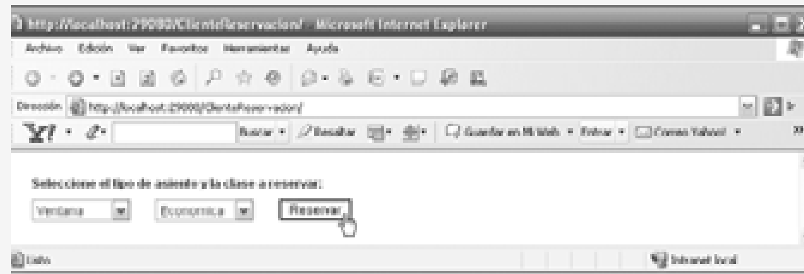


Figura 28.18 | JSP que permite a un usuario seleccionar un asiento. (Parte 3 de 3).

Reservar.java

La figura 28.19 contiene el código de bean de página que proporciona la lógica para `Reservar.jsp` (para ahorrar espacio, no mostramos el código que se genera automáticamente en las líneas 28 a 283). Como vimos en la sección 26.5.2, la clase que representa al bean de página extiende a `AbstractPageBean`. Cuando el usuario hace clic en **Reservar** en la JSP, se ejecuta el manejador de eventos `reservarBoton_action` (líneas 285 a 319). En la línea 289 se crea un objeto proxy `ServicioReservacion1Client`. En las líneas 290 a 292 se utiliza este objeto para invocar el método `reservar` del servicio Web, al cual se le pasa el tipo de asiento seleccionado y el tipo de clase como argumentos. Si `reservar` devuelve `true`, en las líneas 296 a 301 se ocultan los componentes de la GUI en la JSP y se muestra el componente `exitoEtiqueta` (línea 300) para agradecer al usuario por hacer una reservación; en caso contrario, en las líneas 305 a 310 se asegura que los componentes de la GUI se sigan mostrando en pantalla y se muestra el componente `errorEtiqueta` (línea 310) para notificar al usuario que el tipo de asiento solicitado no está disponible, y pide al usuario que intente de nuevo. Cuando el usuario selecciona un valor en uno de los componentes `ListaDesplegable`, se hace una llamada al manejador de eventos correspondiente (`tipoAsientoListaDesplegable_processValueChange` en las líneas 322 a 327, o `claseListaDesplegable_processValueChange` en las líneas 330 a 335) para establecer las propiedades `tipoAsiento` y `tipoClase` de la sesión, las cuales agregamos al bean de sesión de la aplicación. Los valores de estas propiedades se utilizan como argumentos en la llamada al método `reservar` del servicio Web.

```
1  // Fig. 28.19: Reservar.java
2  // Clase de bean de soporte con ámbito de página para el cliente de reservación de asientos
3  package com.deitel.jhtp7.cap28.clientereservacion;
4
5  import com.sun.rave.web.ui.appbase.AbstractPageBean;
6  import com.sun.rave.web.ui.component.Body;
7  import com.sun.rave.web.ui.component.Form;
8  import com.sun.rave.web.ui.component.Head;
9  import com.sun.rave.web.ui.component.Html;
10 import com.sun.rave.web.ui.component.Link;
11 import com.sun.rave.web.ui.component.Page;
12 import javax.faces.FacesException;
13 import com.sun.rave.web.ui.component.Label;
14 import com.sun.rave.web.ui.component.DropDown;
15 import com.sun.rave.web.ui.model.SingleSelectOptionsList;
16 import com.sun.rave.web.ui.component.Button;
17 import com.sun.rave.web.ui.component.StaticText;
18 import webservice.servicioreservacion.
19    servicioreservacion1.ServicioReservacion1Client;
20 import javax.faces.event.ValueChangeEvent;
21
```

Figura 28.19 | Clase de bean de soporte con ámbito de página para el cliente de reservación de asientos. (Parte 1 de 2).

```
public class Reservar extends AbstractPageBean
{
   // Para ahorrar espacio, no mostramos aquí las líneas 24 a 283 del código generado
   // por Java Studio Creator 2. En el archivo Reservar.java podrá ver el código completo
   // junto con los ejemplos de este capítulo.

   // método que invoca al servicio Web cuando el usuario hace clic en el botón Reservar
   public String reservarBoton_action()
   {
      try
      {
         ServicioReservacion1Client cliente = getServicioReservacion1Client1();
         boolean reservado =
            cliente.reservar( getSessionBean1().getTipoAsiento(),
               getSessionBean1().getTipoClase() );

         if ( reservado )
         {
            instruccionesEtiqueta.setVisible( false );
            tipoAsientoListaDesplegable.setVisible( false );
            claseListaDesplegable.setVisible( false );
            reservarBoton.setVisible( false );
            exitoEtiqueta.setVisible( true );
            errorEtiqueta.setVisible( false );
         } // fin de if
         else
         {
            instruccionesEtiqueta.setVisible( true );
            tipoAsientoListaDesplegable.setVisible( true );
            claseListaDesplegable.setVisible( true );
            reservarBoton.setVisible( true );
            exitoEtiqueta.setVisible( false );
            errorEtiqueta.setVisible( true );
         } // fin de else
      } // fin de try
      catch ( Exception e )
      {
         e.printStackTrace();
      } // fin de catch

      return null;
   } // fin del método reservarBoton_action

   // almacena el tipo de asiento seleccionado en el bean de sesión
   public void tipoAsientoListaDesplegable_processValueChange(
      ValueChangeEvent event)
   {
      getSessionBean1().setTipoAsiento(
         ( String ) tipoAsientoListaDesplegable.getSelected() );
   } // fin del método tipoAsientoListaDesplegable_processValueChange

   // almacena la clase seleccionada en el bean de sesión
   public void claseListaDesplegable_processValueChange(
      ValueChangeEvent event)
   {
      getSessionBean1().setTipoClase(
         ( String ) claseListaDesplegable.getSelected() );
   } //fin del método claseListaDesplegable_processValueChange
} // fin de la clase Reservar
```

Figura 28.19 | Clase de bean de soporte con ámbito de página para el cliente de reservación de asientos. (Parte 2 de 2).

28.8 Cómo pasar un objeto de un tipo definido por el usuario a un servicio Web

Los métodos Web que hemos demostrado hasta ahora reciben y devuelven sólo valores de tipos primitivos u objetos `String`. Los servicios Web también pueden recibir y devolver objetos de tipos definidos por el usuario; a éstos se les conoce como **tipos personalizados**. En esta sección presentaremos un servicio Web llamado `GeneradorEcuaciones`, el cual genera preguntas aritméticas aleatorias de tipo `Ecuacion`. El cliente es una aplicación de escritorio para enseñar matemáticas, en la cual el usuario selecciona el tipo de pregunta matemática que desea resolver (suma, resta o multiplicación) y el nivel de habilidad del usuario; el nivel 1 utiliza números de un dígito en cada pregunta, el nivel 2 utiliza números de dos dígitos y el nivel 3 utiliza números de tres dígitos. El cliente pasa esta información al servicio Web, el cual a su vez genera un objeto `Ecuacion` que consiste en números aleatorios con el número apropiado de dígitos. La aplicación cliente recibe el objeto `Ecuacion`, muestra la pregunta de ejemplo al usuario en una aplicación Java, permite al usuario proporcionar una respuesta y la verifica para determinar si es correcta o no.

Serialización de los tipos definidos por el usuario

Anteriormente mencionamos que todos los tipos que se pasan o reciben de servicios Web SOAP deben recibir soporte de SOAP. Entonces, ¿cómo puede SOAP dar soporte a un tipo que no se ha creado todavía? Los tipos personalizados que se envían o reciben de un servicio Web se serializan en formato XML. A este proceso se le conoce como **serialización XML**. El proceso de serializar objetos a XML y deserializar objetos de XML se maneja de manera automática, sin necesidad de que el programador intervenga.

Requerimientos para los tipos definidos por el usuario que se utilizan con métodos Web

Una clase que se utiliza para especificar tipos de parámetros o de valores de retorno en los métodos Web debe cumplir varios requerimientos:

1. Debe proporcionar un constructor predeterminado `public` o sin argumentos. Cuando un servicio Web, o el consumidor de un servicio Web, recibe un objeto serializado XML, el Marco de trabajo JAX-WS 2.0 debe tener la capacidad de llamar a este constructor al momento de deserializar el objeto (es decir, convertirlo de XML otra vez en un objeto Java).
2. Las variables de instancia que deben serializarse en formato XML deben tener métodos *set (establecer)* y *get (obtener)* `public` para acceder a las variables de instancia `private` (recomendado), o las variables de instancia deben declararse `public` (no se recomienda).
3. Las variables de instancia que no sean `public` y deban serializarse deben proporcionar métodos *set* y *get* (incluso si tienen cuerpos vacíos); en caso contrario, no se serializan.

Cualquier variable de instancia que no se serialice simplemente recibe su valor predeterminado (o el valor proporcionado por el constructor sin argumentos) cuando se deserializa un objeto de la clase.



Error común de programación 28.3

Si tratamos de deserializar un objeto de una clase que no tenga un constructor predeterminado o sin argumentos, se produce un error en tiempo de ejecución.

Definición de la clase `Ecuacion`

En la figura 28.20 definimos la clase `Ecuacion`. En las líneas 18 a 31 se define un constructor que recibe tres argumentos: dos valores `int` que representan a los operandos izquierdo y derecho, y un valor `String` que representa la operación aritmética a realizar. El constructor establece las variables de instancia `operandoIzq`, `operandoDer` y `tipoOperacion`, y después calcula el resultado apropiado. El constructor sin argumentos (líneas 13 a 16) llama al constructor de tres argumentos (líneas 18 a 31) y le pasa valores predeterminados. No utilizamos constructor sin argumentos de manera explícita, pero el mecanismo de serialización de XML lo utiliza al momento de deserializar objetos de esta clase. Como proporcionamos un constructor con parámetros, debemos definir de manera explícita el constructor sin argumentos en esta clase, de manera que puedan pasarse (o devolverse) objetos de la clase hacia o desde los métodos Web.

```
// Fig. 28.20: Ecuacion.java
// Clase Ecuacion que contiene información acerca de una ecuación
package com.deitel.jhtp7.cap28.generadorecuaciones;

public class Ecuacion
{
   private int operandoIzq;
   private int operandoDer;
   private int valorResultado;
   private String tipoOperacion;

   // constructor sin argumentos requerido
   public Ecuacion()
   {
      this( 0, 0, "+" );
   } // fin del constructor sin argumentos

   public Ecuacion( int valorIzq, int valorDer, String tipo )
   {
      operandoIzq = valorIzq;
      operandoDer = valorDer;
      tipoOperacion = tipo;

      // determina valorResultado
      if ( tipoOperacion.equals( "+" ) ) // suma
         valorResultado = operandoIzq + operandoDer;
      else if ( tipoOperacion.equals( "-" ) ) // resta
         valorResultado = operandoIzq - operandoDer;
      else // multiplicación
         valorResultado = operandoIzq * operandoDer;
   } // fin del constructor con tres argumentos

   // devuelve una representación String de una Ecuacion
   public String toString()
   {
      return operandoIzq + " " + tipoOperacion + " " +
         operandoDer + " = " + valorResultado;
   } // fin del método toString

   // devuelve el lado izquierdo de la ecuación como un objeto String
   public String getLadoIzq()
   {
      return operandoIzq + " " + tipoOperacion + " " + operandoDer;
   } // fin del método getLadoIzq

   // devuelve el lado derecho de la ecuación como un objeto String
   public String getLadoDer()
   {
      return "" + valorResultado;
   } // fin del método getLadoDer

   // obtiene el operandoIzq
   public int getOperandoIzq()
   {
      return operandoIzq;
   } // fin del método getOperandoIzq

   // obtiene el operandoDer
   public int getOperandoDer()
```

Figura 28.20 | Clase `Ecuacion` que contiene información acerca de una ecuación. (Parte 1 de 2).

```
60       {
61          return operandoDer;
62       } // fin del método getOperandoDer
63
64       // obtiene el valorResultado
65       public int getValorRetorno()
66       {
67          return valorResultado;
68       } // fin del método getValorRetorno
69
70       // obtiene el tipoOperacion
71       public String getTipoOperacion()
72       {
73          return tipoOperacion;
74       } // fin del método getTipoOperacion
75
76       // método set requerido
77       public void setLadoIzq( String value )
78       {
79          // cuerpo vacío
80       } // fin del método setLadoIzq
81
82       // método set requerido
83       public void setLadoDer( String value )
84       {
85          // cuerpo vacío
86       } // fin del método setLadoDer
87
88       // método set requerido
89       public void setOperandoIzq( int value )
90       {
91          // cuerpo vacío
92       } // fin del método setOperandoIzq
93
94       // método set requerido
95       public void setOperandoDer( int value )
96       {
97          // cuerpo vacío
98       } // fin del método setOperandoDer
99
100      // método set requerido
101      public void setValorRetorno( int value )
102      {
103         // cuerpo vacío
104      } // fin del método setValorRetorno
105
106      // método set requerido
107      public void setTipoOperacion( String value )
108      {
109         // cuerpo vacío
110      } // fin del método setTipoOperacion
111   } // fin de la clase Ecuacion
```

Figura 28.20 | Clase `Ecuacion` que contiene información acerca de una ecuación. (Parte 2 de 2).

La clase `Ecuacion` define los métodos `getLadoIzq` y `setLadoIzq` (líneas 41 a 44 y 77 a 80); `getLadoDer` y `setLadoDer` (líneas 47 a 50 y 83 a 86); `getOperandoIzq` y `setOperandoIzq` (líneas 53 a 56 y 89 a 92); `getOperandoDer` y `setOperandoDer` (líneas 59 a 62 y 95 a 98); `getValorRetorno` y `setValorRetorno` (líneas 65 a 68 y 101 a 104); y `getTipoOperacion` y `setTipoOperacion` (líneas 71 a 74 y 107 a 110). El cliente del servicio Web no necesita modificar los valores de las variables de instancia. Sin embargo, recuerde que una propiedad se

puede serializar sólo si tiene los métodos de acceso *get* y *set*, o si es `public`. Por lo tanto, proporcionamos métodos *set* con cuerpos vacíos para cada una de las variables de instancia de la clase. El método `getLadoIzq` (líneas 41 a 44) devuelve un objeto `String` que representa todo lo que hay a la izquierda del signo igual (=) en la ecuación, y `getLadoDer` (líneas 47 a 50) devuelve un objeto `String` que representa todo lo que hay a la derecha del signo de igual (=). El método `getOperandoIzq` (líneas 53 a 56) devuelve el entero a la izquierda del operador, y `getOperandoDer` (líneas 59 a 62) obtiene el entero a la derecha del operador. El método `getValorRetorno` (líneas 65 a 68) devuelve la solución a la ecuación, y `getTipoOperacion` (líneas 71 a 74) devuelve el operador en la ecuación. El cliente en este ejemplo no utiliza la propiedad `ladoDer`, pero la incluimos para que los futuros clientes puedan usarla.

Creación del servicio Web `GeneradorEcuaciones`

La figura 28.21 presenta el servicio Web `GeneradorEcuaciones`, el cual crea objetos `Ecuacion` aleatorios y personalizados. Este servicio Web sólo contiene el método `generarEcuacion` (líneas 18 a 31), el cual recibe dos parámetros: la operación matemática (ya sea "+", "-" o "*") y un `int` que representa el nivel de dificultad (1 a 3).

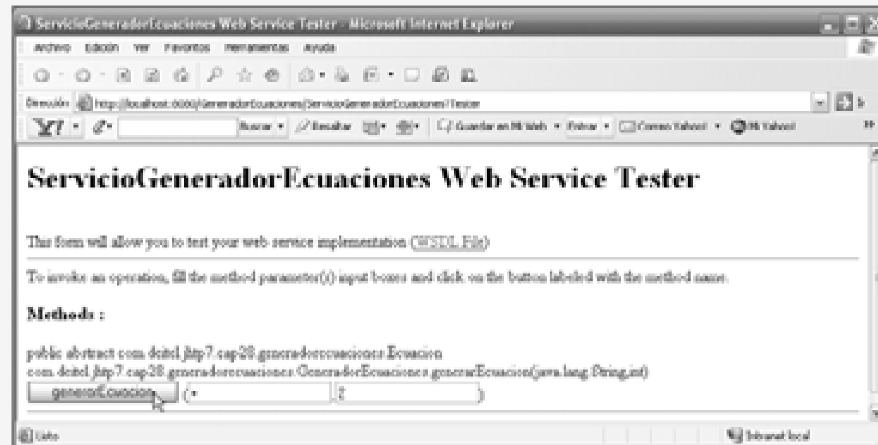
```
1  // Fig. 28.21: GeneradorEcuaciones.java
2  // Servicio Web que genera ecuaciones aleatorias
3  package com.deitel.jhtp7.cap28.generadorecuaciones;
4
5  import java.util.Random;
6  import javax.jws.WebService;
7  import javax.jws.WebMethod;
8  import javax.jws.WebParam;
9
10 @WebService( name = "GeneradorEcuaciones",
11    serviceName = "ServicioGeneradorEcuaciones" )
12 public class GeneradorEcuaciones
13 {
14    private int minimo;
15    private int maximo;
16
17    // genera una ecuación matemática y la devuelve como un objeto Ecuacion
18    @WebMethod( operationName = "generarEcuacion" )
19    public Ecuacion generarEcuacion(
20       @WebParam( name = "operacion" ) String operacion,
21       @WebParam( name = "dificultad" ) int dificultad )
22    {
23       minimo = ( int ) Math.pow( 10, dificultad - 1 );
24       maximo = ( int ) Math.pow( 10, dificultad );
25
26       Random objetoAleatorio = new Random();
27
28       return new Ecuacion(
29          objetoAleatorio.nextInt( maximo - minimo ) + minimo,
30          objetoAleatorio.nextInt( maximo - minimo ) + minimo, operacion );
31    } // fin del método generarEcuacion
32 } // fin de la clase GeneradorEcuaciones
```

Figura 28.21 | Servicio Web que genera ecuaciones aleatorias.

Prueba del servicio Web `GeneradorEcuaciones`

En la figura 28.22 se muestra el resultado de probar el servicio Web `GeneradorEcuaciones` con la página Web `Tester`. En la *parte (b)* de la figura, observe que el valor de retorno de nuestro método Web está codificado en XML. Sin embargo, este ejemplo es distinto de los anteriores porque el XML especifica los valores para todos los datos del objeto serializado en XML que se devuelve. La clase proxy recibe este valor de retorno y lo deserializa en un objeto de la clase `Ecuacion`, y después lo pasa al cliente.

a) Uso de la página Web del servicio Web `GeneradorEcuaciones` para generar un objeto `Ecuacion`.



b) Resultado de generar un objeto `Ecuacion`.

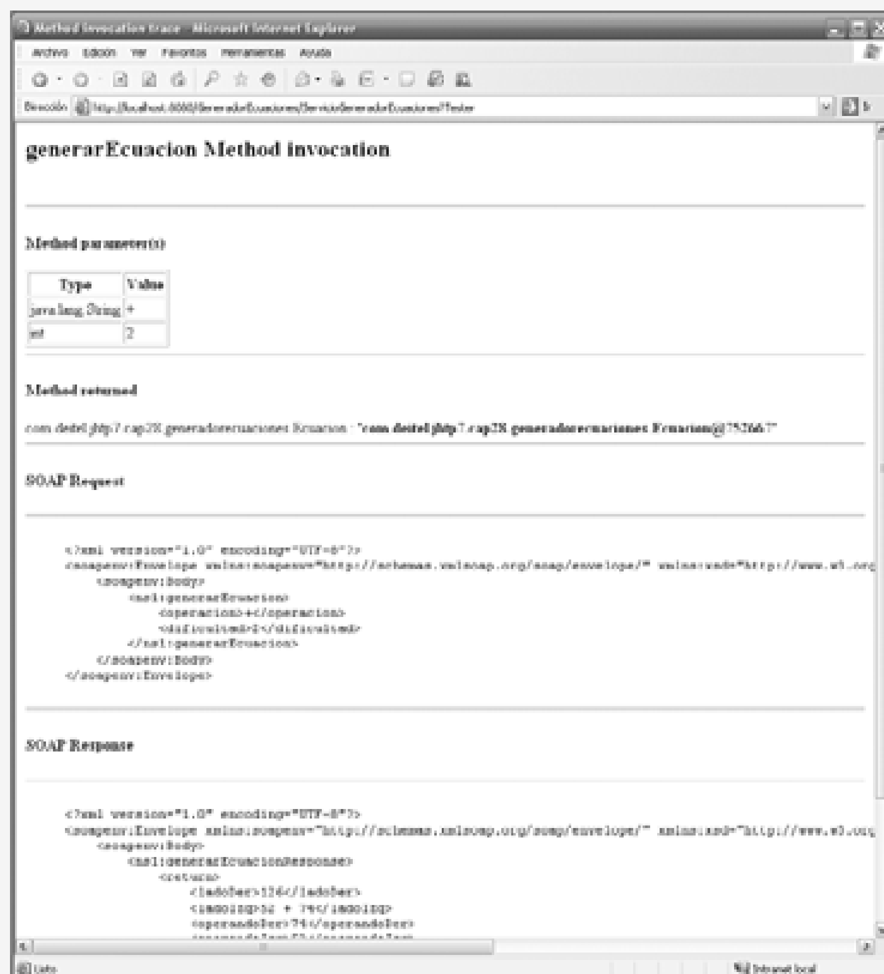


Figura 28.22 | Prueba de un método Web que devuelve un objeto `Ecuacion` serializado con XML.

Observe que *no* se está pasando un objeto `Ecuacion` entre el servicio Web y el cliente. En vez de ello, la información en el objeto se envía en forma de datos codificados en XML. Los clientes creados en Java tomarán la información y crearán un nuevo objeto `Ecuacion`. Sin embargo, los clientes creados en otras plataformas tal vez utilicen la información en forma distinta. Los lectores que creen clientes en otras plataformas deben revisar la documentación de los servicios Web para la plataforma específica que utilicen, para ver cómo pueden sus clientes procesar tipos personalizados.

Detalles del servicio Web `GeneradorEcuaciones`

Vamos a analizar el método Web `GenerarEcuacion` más de cerca. En las líneas 23 a 24 de la figura 28.21 se definen los límites superior e inferior de los números aleatorios que utiliza el método para generar un objeto `Ecuacion`. Para establecer estos límites, el programa llama primero al método `static pow` de la clase `Math`; este método eleva su primer argumento a la potencia de su segundo argumento. El valor de la variable `minimo` se determina elevando `10` a una potencia que sea uno menos que `nivel` (línea 23). Esto calcula el número más pequeño con `nivel` dígitos. Si `nivel` es `1`, `minimo` es `1`; si `nivel` es `2`, `minimo` es `10`; y si `nivel` es `3`, `minimo` es `100`. Para calcular el valor de `maximo` (el límite superior para cualquier número generado al azar que se utilice para formar un objeto `Ecuacion`), el programa eleva `10` a la potencia del argumento `nivel` especificado (línea 23). Si `nivel` es `1`, `maximo` es `10`; si `nivel` es `2`, `maximo` es `100`; y si `nivel` es `3`, `maximo` es `1000`.

En las líneas 28 a 30 se crea y devuelve un nuevo objeto `Ecuacion`, el cual consiste en dos números aleatorios y el objeto `String` llamado `operacion` que recibe `generarEcuacion`. El programa llama al método `Random nextInt`, el cual devuelve un `int` que es menor que el límite superior especificado. Este método genera un valor de operando izquierdo que es mayor o igual a `minimo`, pero menor que `maximo` (es decir, un número con `nivel` dígitos). El operando derecho es otro número aleatorio con las mismas características.

Consumo del servicio Web `GeneradorEcuaciones`

La aplicación **Tutor de matemáticas** (figura 28.23) utiliza el servicio Web `GeneradorEcuaciones`. Esta aplicación llama al método `generarEcuacion` del servicio Web para crear un objeto `Ecuacion`. Después, el tutor muestra el lado izquierdo del objeto `Ecuacion` y espera a que el usuario introduzca datos. En la línea 9 también se declara una variable de instancia llamada `ServicioGeneradorEcuaciones`, la cual utilizamos para obtener un objeto proxy `GeneradorEcuaciones`. En las líneas 10 a 11 se declaran variables de instancia de los tipos `GeneradorEcuaciones` y `Ecuacion`.

Después de mostrar una ecuación, la aplicación espera a que el usuario escriba una respuesta. La opción predeterminada para el nivel de dificultad es **Números de un dígito**, pero el usuario puede cambiar esto si selecciona un nivel del objeto `JComboBox` llamado **Seleccione el nivel**. Al hacer clic en cualquiera de los niveles se invoca el método `nivelJComboBoxItemStateChanged` (líneas 158 a 163), el cual establece la variable `dificultad` con el nivel seleccionado por el usuario. Aunque la opción predeterminada para el tipo de pregunta es **Suma**, el usuario también puede cambiar esto si selecciona una operación del objeto `JComboBox` **Seleccione la operación**. Al hacer esto, se invoca al método `operacionJComboBoxItemStateChanged` (líneas 166 a 177), el cual establece la variable `String` llamada `operacion` con el símbolo matemático apropiado.

Cuando el usuario hace clic en el objeto `JButton` **Generar ecuación**, el método `generarJButtonActionPerformed` (líneas 207 a 221) invoca al método `generarEcuacion` (línea 212) del servicio Web `GeneradorEcuaciones`. Después de recibir un objeto `Ecuacion` del servicio Web, el manejador muestra el lado izquierdo de la ecuación en el componente `ecuacionJLabel` (línea 214) y habilita el component `comprobarRespuestaJButton`, de manera que el usuario pueda enviar una respuesta. Cuando el usuario hace clic en el botón `JButton` **Comprobar respuesta**, el método `comprobarRespuestaJButtonActionPerformed` (líneas 180 a 204) determina si el usuario proporcionó la respuesta correcta.

```
1  // Fig. 28.23: ClienteGeneradorEcuacionesJFrame.java
2  // Programa tutor de matemáticas que usa servicios Web para generar ecuaciones
3  package com.deitel.jhtp7.cap28.clientegeneradorecuaciones;
4
5  import javax.swing.JOptionPane;
6
```

Figura 28.23 | Aplicación tutor de matemáticas. (Parte 1 de 4).

```
public class ClienteGeneradorEcuacionesJFrame extends javax.swing.JFrame
{
   private ServicioGeneradorEcuaciones servicio; // se utiliza para obtener el proxy
   private GeneradorEcuaciones proxy; // se utiliza para acceder al servicio Web
   private Ecuacion ecuacion; // representa una ecuación
   private int respuesta; // la respuesta del usuario a la pregunta
   private String operacion = "+"; // operación matemática +, - o *
   private int dificultad = 1; // 1, 2 o 3 dígitos en cada número

   // constructor sin argumentos
   public ClienteGeneradorEcuacionesJFrame()
   {
      initComponents();

      try
      {
         // crea los objetos para acceder al servicio GeneradorEcuaciones
         servicio = new ServicioGeneradorEcuaciones();
         proxy = servicio.getGeneradorEcuacionesPort();
      } // fin de try
      catch ( Exception ex )
      {
         ex.printStackTrace();
      } // fin de catch
   } // fin de constructores sin argumentos

   // El método initComponents se genera automáticamente por Netbeans y se llama
   // desde el constructor para inicializar la GUI. Aquí no se muestra este
   // método para ahorrar espacio. Abra ClienteGeneradorEcuacionesJFrame.java en la
   // carpeta de este ejemplo para ver el código generado completo (líneas 37 a 156).

   // obtiene el nivel de dificultad seleccionado por el usuario
   private void nivelJComboBoxItemStateChanged(
      java.awt.event.ItemEvent evt )
   {
      // los índices empiezan en 0, por lo que se suma 1 para obtener el nivel de dificultad
      dificultad = nivelJComboBox.getSelectedIndex() + 1;
   } fin del método nivel JComboBoxItemStateChanged

   // obtiene la operación matemática seleccionada por el usuario
   private void operacionJComboBoxItemStateChanged(
      java.awt.event.ItemEvent evt )
   {
      String elemento = ( String ) operacionJComboBox.getSelectedItem();

      if ( elemento.equals( "Suma" ) )
         operacion = "+"; // el usuario seleccionó suma
      else if ( elemento.equals( "Resta" ) )
         operacion = "-"; // el usuario seleccionó resta
      else
         operacion = "*"; // el usuario seleccionó multiplicación
   } // fin del método operationJComboBoxItemStateChanged

   // comprueba la respuesta del usuario
   private void comprobarRespuestaJButtonActionPerformed(
      java.awt.event.ActionEvent evt )
   {
      if ( respuestaJTextField.getText().equals( "" ) )
```

Figura 28.23 | Aplicación tutor de matemáticas. (Parte 2 de 4).

```
      {
         JOptionPane.showMessageDialog(
            this, "Escriba su respuesta." );
      } // fin de if

      int respuestaUsuario = Integer.parseInt( respuestaJTextField.getText() );

      if ( respuestaUsuario == respuesta )
      {
         ecuacionJLabel.setText( "" );
         respuestaJTextField.setText( "" );
         comprobarRespuestaJButton.setEnabled( false );
         JOptionPane.showMessageDialog( this, "Correcto! Bien hecho!",
            "Correcto", JOptionPane.PLAIN_MESSAGE );
      } // fin de if
      else
      {
         JOptionPane.showMessageDialog( this, "Incorrecto. Intente de nuevo.",
            "Incorrecto", JOptionPane.PLAIN_MESSAGE );
      } // fin de else
   } // fin del método checkAnswerJButtonActionPerformed

   // genera un nuevo objeto Ecuacion con base en las selecciones del usuario
   private void generarJButtonActionPerformed(
      java.awt.event.ActionEvent evt )
   {
      try
      {
         ecuacion = proxy.generarEcuacion( operacion, dificultad );
         respuesta = ecuacion.getValorRetorno();
         ecuacionJLabel.setText( ecuacion.getLadoIzq() + " = " );
         comprobarRespuestaJButton.setEnabled( true );
      } // fin de try
      catch ( Exception e )
      {
         e.printStackTrace();
      } // fin de catch
   } // fin del método generateJButtonActionPerformed

   // empieza la ejecución del programa
   public static void main( String args[] )
   {
      java.awt.EventQueue.invokeLater(
         new Runnable()
         {
            public void run()
            {
               new ClienteGeneradorEcuacionesJFrame().setVisible( true );
            } // fin del método run
         } // fin de la clase interna anónima
      ); // fin de la llamada a java.awt.EventQueue.invokeLater
   } // fin del método main

   // Variables declaration - do not modify
   private javax.swing.JButton comprobarRespuestaJButton;
   private javax.swing.JLabel ecuacionJLabel;
   private javax.swing.JButton generarJButton;
   private javax.swing.JComboBox nivelJComboBox;
   private javax.swing.JLabel nivelJLabel;
```

Figura 28.23 | Aplicación tutor de matemáticas. (Parte 3 de 4).

```
   private javax.swing.JComboBox operacionJComboBox;
   private javax.swing.JLabel operacionJLabel;
   private javax.swing.JLabel preguntaJLabel;
   private javax.swing.JLabel respuestaJLabel;
   private javax.swing.JTextField respuestaJTextField;
   // Fin de variables declaration
} // fin de la clase ClienteGeneradorEcuacionesJFrame
```



Figura 28.23 | Aplicación tutor de matemáticas. (Parte 4 de 4).

28.9 Conclusión

En este capítulo se introdujeron los servicios Web JAX-WS 2.0, los cuales promueven la portabilidad y reutilización de software en aplicaciones que operan a través de Internet. Aprendió que un servicio Web es un componente de software almacenado en una computadora a la que una aplicación (u otro componente de software) puede acceder en otra computadora a través de una red, y se comunican a través de tecnologías como XML, SOAP y

HTTP. Hablamos sobre los diversos beneficios de este tipo de computación distribuida; por ejemplo, los clientes pueden acceder a los datos en equipos remotos, los clientes que no tengan el poder de procesamiento para realizar cálculos específicos pueden aprovechar los recursos de los equipos remotos, y pueden desarrollarse por completo tipos nuevos de aplicaciones innovadoras.

Explicamos cómo Netbeans, Sun Java Studio Creator 2 y las APIs de JAX-WS 2.0 facilitan la creación y el consumo de servicios Web. Le mostramos cómo establecer proyectos y archivos en estas herramientas, y cómo las herramientas administran la infraestructura del servicio Web necesaria para dar soporte a los servicios creados por el programador. Aprendió a definir los servicios Web y los métodos Web, así como a consumirlos desde aplicaciones Java de escritorio creadas en Netbeans, y también desde aplicaciones Web creadas en Sun Java Studio Creator 2. Después de explicar la mecánica de los servicios Web con nuestro ejemplo `EnteroEnorme`, demostramos servicios Web más sofisticados que utilizan rastreo de sesiones tanto del lado servidor como del lado cliente, y servicios Web que acceden a bases de datos mediante el uso de JDBC. También explicamos la serialización con XML y le mostramos cómo pasar objetos de tipos definidos por el usuario a los servicios Web, y cómo devolverlos de los servicios Web.

En el siguiente capítulo hablaremos acerca de cómo dar formato a la salida con el método `System.out.printf` y la clase `Formatter`.

28.10 Recursos Web

Además de los recursos Web que se muestran a continuación, también puede consultar los recursos Web relacionados con JSP que se proporcionan al final del capítulo 26.

`www.deitel.com/WebServices/`

Visite nuestro Centro de recursos de servicios Web para obtener información acerca de cómo diseñar e implementar servicios Web en muchos lenguajes, e información acerca de los servicios Web ofrecidos por compañías como Google, Amazon y eBay. También encontrará muchas herramientas adicionales de Java para publicar y consumir servicios Web.

`www.deitel.com/java/`
`www.deitel.com/JavaSE6Mustang/`
`www.deitel.com/JavaEE5/`
`www.deitel.com/JavaCertification/`
`www.deitel.com/JavaDesignPatterns/`

Nuestros Centros de recursos sobre Java proporcionan información específica de este lenguaje, como libros, documentos, artículos, diarios, sitios Web y blogs que abarcan una gran variedad de temas relacionados con Java (incluyendo los servicios Web de java).

`www.deitel.com/ResourceCenters.html`

De un vistazo a nuestra lista cada vez más extensa de Centros de recursos sobre programación, Web 2.0, software y demás temas interesantes.

`java.sun.com/webservices/jaxws/index.jsp`

El sitio oficial para la API de Sun Java para los Servicios Web de XML (JAX-WS). Incluye la API, documentación, tutoriales y demás vínculos de utilidad.

`www.webservices.org`

Ofrece noticias relacionadas con la industria, artículos y recursos para los servicios Web.

`www-130.ibm.com/developerworks/webservices`

El sitio de IBM para la arquitectura orientada al servicio (SOA) y los servicios Web incluye artículos, descargas, demos y foros de discusión relacionados con la tecnología de los servicios Web.

`www.w3.org/TR/wsdl`

Ofrece gran cantidad de documentación acerca de WSDL, incluyendo una explicación detallada de los servicios Web y las tecnologías relacionadas, como XML, SOAP, HTTP los tipos MIME en el contexto de WSDL.

`www.w3.org/TR/soap`

Ofrece gran cantidad de documentación acerca de los mensajes SOAP, el uso de SOAP con HTTP y cuestiones de seguridad relacionadas con SOAP.

`www.ws-i.org`

El sitio Web de la Organización de Interoperabilidad de Servicios Web proporciona información detallada sobre la creación de servicios Web basados en estándares que promuevan la interoperabilidad y una verdadera independencia de la plataforma.

`webservices.xml.com/security`

Artículos acerca de la seguridad de los servicios Web y los protocolos de seguridad estándar.

Servicios Web basados en REST

`en.wikipedia.org/wiki/REST`

Recurso de Wikipedia que explica la Transferencia representativa de estado (REST).

`www.xfront.com/REST-Web-Services.html`

Artículo titulado “Building Web Services the REST Way” (Cómo crear servicios Web al estilo REST).

`www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm`

La disertación que propuso originalmente el concepto de los servicios basados en REST.

`rest.blueoxen.net/cgi-bin/wiki.pl?ShortSummaryOfRest`

Una breve introducción a REST (en inglés).

`www.prescod.net/rest`

Vínculos a muchos recursos sobre REST (en inglés).

Resumen

Sección 28.1 Introducción

- Un servicio Web es un componente de software almacenado en una computadora a la que se puede acceder mediante llamadas a métodos desde una aplicación (u otro componente de software) en otra computadora, a través de una red.
- Los servicios Web se comunican mediante el uso de tecnologías como XML y HTTP.
- El Protocolo simple de acceso a objetos (SOAP) es un protocolo basado en XML que permite la comunicación entre los clientes y los servicios Web, en forma independiente de la plataforma.
- Los servicios Web permiten a los comercios realizar transacciones a través de servicios Web estandarizados y ampliamente disponibles, en vez de depender de aplicaciones propietarias.
- Las compañías como Amazon, Google, eBay, PayPal y muchas otras están usando servicios Web para su beneficio, al hacer que sus aplicaciones del lado cliente estén disponibles para sus socios a través de los servicios Web.
- Al comprar servicios Web y utilizar la gran diversidad de servicios Web gratuitos, las compañías pueden invertir menos tiempo en desarrollar nuevas aplicaciones y pueden crear nuevas e innovadoras aplicaciones.
- Netbeans 5.5 y Sun Java Studio Creator 2 (ambos desarrollados por Sun) son dos de las diversas herramientas que permiten a los programadores “pubicar” y “consumir” servicios Web.

Sección 28.2 Fundamentos de los servicios Web de Java

- La computadora en la que reside un servicio Web se conoce como equipo remoto o servidor. Una aplicación cliente que accede a un servicio Web envía una llamada a un método a través de la red a un equipo remoto, el cual procesa la llamada y devuelve una respuesta a la aplicación, a través de la red.
- En Java, un servicio Web se implementa como una clase. La clase que representa el servicio Web reside en un servidor; no forma parte de la aplicación cliente.
- Al proceso de hacer que un servicio Web esté disponible para recibir peticiones de los clientes se le conoce como publicación de un servicio Web; al proceso de usar un servicio Web desde una aplicación cliente se le conoce como consumo de un servicio Web.
- Una aplicación que consume un servicio Web consiste de dos partes: un objeto de una clase proxy para interactuar con el servicio Web y una aplicación cliente que consume el servicio Web, al invocar métodos en el objeto proxy. El objeto proxy maneja los detalles relacionados con la comunicación con el servicio Web por el cliente.
- Las peticiones a (y las respuestas de) los servicios Web creados con JAX-WS 2.0 se transmiten comúnmente mediante SOAP. Cualquier cliente capaz de generar y procesar mensajes SOAP puede interactuar con un servicio Web, sin importar el lenguaje en el que esté escrito.

Sección 28.3.1 Creación de un proyecto de aplicación Web y cómo agregar una clase de servidor Web en Netbeans

- Al crear un servicio Web en Netbeans, nos enfocamos en la lógica del servicio Web y dejamos que el IDE se encargue de la infraestructura del servicio Web.
- Para crear un servicio Web en Netbeans, primero debemos crear un proyecto de tipo **aplicación Web (Web Application)**. Netbeans utiliza este tipo de proyecto para las aplicaciones Web que se ejecutan en clientes basados en navegador, y para los servicios Web invocados por otras aplicaciones.
- Al crear una **aplicación Web** en Netbeans, el IDE genera archivos adicionales que dan soporte a la aplicación Web.

Sección 28.3.2 Definición del servicio Web `EnteroEnorme` en Netbeans

- De manera predeterminada, cada nueva clase de servicio Web que se crea con las APIs de JAX-WS es un POJO (Objeto Java simple); no necesita extender una clase o implementar una interfaz para crear un servicio Web.
- Al compilar una clase que utiliza estas anotaciones de JAX-WS 2.0, el compilador crea el marco de trabajo de código compilado que permite al servicio Web esperar las peticiones de los clientes y responder a esas peticiones.
- La anotación `@WebService` indica que una clase representa a un servicio Web. El elemento opcional `name` especifica el nombre de la clase proxy que se generará para el cliente. El elemento opcional `serviceName` especifica el nombre de la clase que el cliente debe utilizar para obtener un objeto de la clase proxy.
- Netbeans coloca la anotación `@WebService` al principio de cada nueva clase de servicio Web que crea el programador. Se pueden agregar los elementos opcionales `name` y `serviceName` en los paréntesis de la anotación.
- Los métodos que se etiquetan con la anotación `@WebMethod` se pueden llamar en forma remota.
- Los métodos que no están etiquetados con `@WebMethod` no están accesibles para los clientes que consumen el servicio Web. Por lo general, éstos son métodos utilitarios dentro de la clase de servicio Web.
- La anotación `@WebMethod` tiene un elemento `operationName` opcional para especificar el nombre del método que está expuesto al cliente del servicio Web.
- Los parámetros de los métodos Web se anotan con la anotación `@WebParam`. El elemento opcional `name` indica el nombre del parámetro que está expuesto a los clientes del servicio Web.

Sección 28.3.3 Publicación del servicio Web `EnteroEnorme` desde Netbeans

- Netbeans se encarga de todos los detalles relacionados con la generación y despliegue de un servicio Web por nosotros. Esto incluye crear el marco de trabajo requerido para dar soporte al servicio Web.
- Para determinar si hay errores de compilación en el proyecto, haga clic con el botón derecho del ratón en el nombre del proyecto en la ficha **Projects** de Netbeans, y después seleccione la opción **Build Project**.
- Seleccione **Deploy Project** para desplegar el proyecto en el servidor Web que seleccionó durante la configuración de la aplicación.
- Seleccione **Run Project** para ejecutar la aplicación Web.
- Las opciones **Deploy Project** y **Run Project** también generan el proyecto si ha cambiado, e inician el servidor de aplicaciones en caso de que no se encuentre ya en ejecución.
- Para asegurar que se vuelvan a compilar todos los archivos de código fuente en un proyecto durante la siguiente operación de generación, puede usar las opciones **Clean Project** o **Clean and Build Project**.

Sección 28.3.4 Prueba del servicio Web `EnteroEnorme` con la página Web `Tester` de Sun Java System Application Server

- Sun Java System Application Server puede crear en forma dinámica una página Web llamada `Tester` para probar los métodos de un servicio Web desde un navegador Web. Para habilitar esta característica, use las opciones de ejecución (**Run**) del proyecto.
- Para mostrar la página Web `Tester`, ejecute la aplicación desde Netbeans o escriba el URL del servicio Web en el campo dirección del navegador, seguido de `?Tester`.
- Un cliente puede acceder a un servicio Web sólo cuando el servidor de aplicaciones se encuentra en ejecución. Si Netbeans inicia el servidor de aplicaciones por usted, éste se cerrará cuando cierre Netbeans. Para mantener el servidor de aplicaciones en ejecución, puede iniciarlo en forma independiente de Netbeans.

Sección 28.3.5 Descripción de un servicio Web con el Lenguaje de descripción de servicios Web (WSDL)

- Para consumir un servicio Web, un cliente debe saber en dónde encontrar el servicio Web, y debe recibir la descripción del mismo.
- JAX-WS utiliza el Lenguaje de descripción de servicios Web (WSDL): un vocabulario XML estándar para describir servicios Web de manera independiente de la plataforma.

- No necesita comprender el WSDL para aprovechar sus beneficios; el servidor genera un WSDL del servicio Web en forma dinámica por usted, y las herramientas cliente pueden analizar el WSDL para ayudarnos a crear la clase proxy del lado cliente que utiliza un cliente para acceder al servicio Web.
- Como el WSDL se crea en forma dinámica, los clientes siempre reciben la descripción más actualizada de un servicio Web desplegado.
- Para ver el WSDL de un servicio Web, escriba su URL en el campo dirección del navegador seguido de `?WSDL`, o haga clic en el vínculo **WSDL File** en la página Web `Tester` de Sun Java System Application Server.

Sección 28.4 Cómo consumir un servicio Web

- El cliente de un servicio Web puede ser cualquier tipo de aplicación, o incluso otro servicio Web.
- En Netbeans, para habilitar una aplicación cliente de manera que pueda consumir un servicio Web, hay que agregar una referencia del servicio Web a la aplicación, la cual define la clase proxy del lado cliente.

Sección 28.4.1 Creación de un cliente para consumir el servicio Web `EnteroEnorme`

- Al agregar una referencia de servicio Web, el IDE crea y compila los artefactos del lado cliente: el marco de trabajo de código de Java que da soporte a la clase proxy del lado cliente.
- El cliente llama a los métodos en un objeto proxy, el cual usa los artefactos del lado cliente para interactuar con el servicio Web.
- Para agregar una referencia de servicio Web, haga clic con el botón derecho del ratón en el nombre del proyecto del cliente en la ficha **Projects** de Netbeans, y después seleccione **New > Web Service Client…** En el campo **WSDL URL** del cuadro de diálogo, especifique el URL del WSDL del servicio Web.
- Netbeans utiliza la descripción WSDL para generar la clase proxy y los artefactos del lado cliente.
- Al especificar el servicio Web que el programador desea consumir, Netbeans copia el WSDL del servicio Web en un archivo en el proyecto. Podemos ver este archivo desde la ficha **Files** de Netbeans, expandiendo los nodos en la carpeta `xml-resources` del proyecto.
- Los artefactos del lado cliente y la copia del cliente del archivo WSDL se pueden regenerar, haciendo clic con el botón derecho del ratón en el nodo del servicio Web en la ficha **Projects** de Netbean, y seleccionando **Refresh Client**.
- Para ver los artefactos del lado cliente generados por el IDE, seleccione la ficha **Files** de Netbeans y expanda la carpeta `build` del proyecto.

Sección 28.5 SOAP

- SOAP es un protocolo basado en XML, de uso común e independiente de la plataforma, que facilita las llamadas a procedimientos remotos, comúnmente a través de HTTP.
- El protocolo que transmite mensajes de petición y respuesta también se conoce como el formato de cable o protocolo de cable del servicio Web, ya que define la forma en que se envía la información "a lo largo del cable".
- Cada petición y respuesta se empaquetan en un mensaje SOAP (también conocido como envoltura SOAP), el cual contiene la información que un servicio Web requiere para procesar el mensaje.
- El formato de cable utilizado para transmitir peticiones y respuestas debe tener soporte para todos los tipos que se pasen de una aplicación a otra. SOAP soporta los tipos primitivos y sus tipos de envoltura, así como `Date`, `Time` y otros. SOAP también puede transmitir arreglos y objetos de tipos definidos por el usuario.
- Cuando un programa invoca a un método Web, la petición y toda la información relevante se empaquetan en un mensaje SOAP y se envían al servidor en el que reside el servicio Web. El servicio Web procesa el contenido del mensaje SOAP; este mensaje especifíca el método a invocar y sus argumentos. Una vez que el servicio Web recibe y analiza una petición, se hace una llamada al método apropiado y la respuesta se envía de vuelta al cliente, en otro mensaje SOAP. El proxy del lado cliente analiza la respuesta, que contiene el resultado de la llamada al método, y después devuelve el resultado a la aplicación cliente.
- Los mensajes SOAP se generan de manera automática por usted. Por lo tanto, no necesita comprender los detalles acerca de SOAP o XML para aprovechar estas tecnologías al publicar y consumir servicios Web.

Sección 28.6 Rastreo de sesiones en los servicios Web

- Puede ser benéfico para un servicio Web mantener la información de estado del cliente, con lo cual se elimina la necesidad de pasar la información del cliente entre éste y el servicio Web varias veces. Al almacenar la información de la sesión, el servicio Web puede también diferenciar a un cliente de otro.

Sección 28.6.1 Creación de un servicio Web `Blackjack`

- Para usar el rastreo de sesiones en un servicio Web, debemos incluir código para los recursos que mantienen la información de estado de la sesión. Anteriormente, había que escribir el código, que algunas veces era tedioso, para

crear estos recursos. Sin embargo, JAX-WS se encarga de esto por nosotros mediante la anotación `@Resource`. Esta anotación permite a herramientas como Netbeans "inyectar" el código complejo de soporte en nuestra clase, con lo cual nos podemos enfocar en la lógica de negocios, en vez de hacerlo en el código de soporte.
- Al uso de anotaciones para agregar código que dé soporte a nuestras clases se le conoce como inyección de dependencias. Las anotaciones como `@WebService`, `@WebMethod` y `@WebParam` también realizan la inyección de dependencias.
- Un objeto `WebServiceContext` permite a un servicio Web acceder a la información para una petición específica y darle mantenimiento, como el estado de la sesión. El código necesario que crea a un objeto `WebServiceContext` se inyecta en la clase mediante la anotación `@Resource`.
- El objeto `WebServiceContext` se utiliza para obtener un objeto `MessageContext`. Un servicio Web utiliza a un objeto `MessageContext` para obtener un objeto `HttpSession` para el cliente actual.
- El método `get` del objeto `MessageContext` se utiliza para obtener el objeto `HttpSession` para el cliente actual. El método `get` recibe una constante que indica lo que se debe obtener del objeto `MessageContext`. La constante `MessageContext.SERVLET_REQUEST` indica que deseamos obtener el objeto `HttpServletRequest` para el cliente actual. Después llamamos al método `getSession` para obtener el objeto `HttpSession` del objeto `HttpServletRequest`.
- El método `getAttribute` de `HttpSession` recibe un objeto `String` que identifica el objeto `Object` a obtener del estado de la sesión.

Sección 28.6.2 Cómo consumir el servicio Web `Blackjack`

- En el marco de trabajo JAX-WS 2.0, el cliente debe indicar si desea permitir al servicio Web mantener la información de la sesión. Para ello, primero convertimos el objeto proxy al tipo de interfaz `BindingProvider`. Un objeto `BindingProvider` permite al cliente manipular la información de petición que se enviará al servidor. Esta información se almacena en un objeto que implementa a la interfaz `RequestContext`. Los objetos `BindingProvider` y `RequestContext` son parte del marco de trabajo que crea el IDE cuando agregamos un cliente de servicio Web a la aplicación.
- A continuación, se invoca el método `getRequestContext` de `BindingProvider` para obtener el objeto `RequestContext`. Luego se hace una llamada al método `put` de `RequestContext` para establecer la propiedad `BindingProvider.SESSION_MAINTAIN_PROPERTY` a `true`, lo cual permite el rastreo de sesiones desde el lado cliente, de manera que el servicio Web sepa qué cliente está invocando a sus métodos.

Sección 28.7.1 Configuración de Java DB en Netbeans y creación de la base de datos `Reservacion`

- Para agregar un servidor de bases de datos Java DB en Netbeans, realice los siguientes pasos: Seleccione **Tools > Options...** para que aparezca el cuadro de diálogo **Options** de Netbeans. Haga clic en el botón **Advanced Options** para mostrar el cuadro de diálogo **Advanced Options**. En **IDE Configuration**, expanda el nodo **Server and External Tool Settings** y seleccione **Java DB Database**. Si las propiedades de Java DB no están ya configuradas, establezca la propiedad **Java DB Location** con la ubicación de Java DB en su sistema. Además, establezca la propiedad **Database Location** con la ubicación en donde desea que se almacenen las bases de datos Java DB.
- Para crear una nueva base de datos: seleccione **Tools > Java DB Datables > Create Java DB Database...**. Escriba el nombre de la base de datos a crear, un nombre de usuario y contraseña, y después haga clic en **OK** para crear la base de datos.
- Puede utilizar la ficha **Runtime** de Netbeans para crear tablas y ejecutar instrucciones SQL que llenen la base de datos con información. Haga clic en la ficha **Runtime** de Netbeans y expanda el nodo **Databases**.
- Netbeans debe estar conectado a la base de datos para ejecutar instrucciones SQL. Si Netbeans no está conectado a la base de datos, aparecerá el ícono enseguida del URL de la base de datos. En este caso, haga clic con el botón derecho del ratón en el ícono y seleccione **Connect...**. Una vez conectado, el ícono cambiará a .

Sección 28.7.2 Creación de una aplicación Web para interactuar con el servicio Web `Reservacion`

- Para agregar un servicio Web a una aplicación Web en Java Studio Creator 2, haga clic en el botón **Agregar servicio Web...** ( ). Después podrá especificar el WSDL del servicio Web en el cuadro de diálogo que aparezca.

Sección 28.8 Cómo pasar un objeto de un tipo definido por el usuario a un servicio Web

- Los servicios Web pueden recibir y devolver objetos de tipos definidos por el usuario; a éstos se les conoce como tipos personalizados.
- Los tipos personalizados que se envían o reciben de un servicio Web mediante el uso de SOAP se serializan en formato XML. A este proceso se le conoce como serialización XML y se maneja de manera automática, sin necesidad de que el programador intervenga.

- Una clase que se utiliza para especificar tipos de parámetros o de valores de retorno en los métodos Web debe proporcionar un constructor predeterminado `public` o sin argumentos. Además, las variables de instancia que deban deserializarse deben tener métodos *set* y *get* `public`, o las variables de instancia se deben declarar como `public`.
- Cualquier variable de instancia que no se serialice, simplemente recibe su valor predeterminado (o el valor proporcionado por el constructor sin argumentos) a la hora de deserializar un objeto de la clase.

Terminología

`@Resource`, anotación
`@WebMethod`, anotación
`@WebParam`, anotación
`@WebService`, anotación
`AbstractPageBean`, clase
Add Server Instance, cuadro de diálogo
agregar una referencia de servicio Web a un proyecto en Netbeans
analizar un mensaje SOAP
artefactos del lado cliente
artefactos del lado servidor
BEA Weblogic Server
`BindingProvider`, interfaz
Build Project, opción en Netbeans
clase proxy para un servicio Web
Clean and Build Project, opción en Netbeans
Clean Project, opción en Netbeans
consumir un servicio Web
Deploy Project, opción en Netbeans
desplegar un servicio Web
envoltura SOAP
equipo remoto
formato de cable
`get`, método de la interfaz `MessageContext`
`getRequestContext`, método de la interfaz `BindingProvider`
IDE Netbeans 5.5
inyección de dependencias
JAX-WS 2.0
JBoss Application Server
Lenguaje de descripción de servicios Web (WSDL)
mensaje SOAP
`MessageContext`, interfaz
`name`, elemento de la anotación `@WebParam`
`name`, elemento de la anotación `@WebService`
New Project, cuadro de diálogo en Netbeans
New Web Service Client, cuadro de diálogo en Netbeans
New Web Service, cuadro de diálogo en Netbeans
objeto proxy maneja los detalles de la comunicación con el servicio Web
`operationName`, elemento de la anotación `@WebMethod`
Organización de Interoperabilidad de Servicios Web (WS-I)
POJO (Objeto Java simple)
probar un servicio Web
Project Properties, cuadro de diálogo en Netbeans
protocolo de cable
publicar un servicio Web
`put`, método de la interfaz `RequestContext`
rastreo de sesiones en servicios Web
referencia de servicio Web
`RequestContext`, interfaz
REST (Transferencia representativa de estado)
Run Project, opción en Netbeans
serialización en XML
Server Manager, cuadro de diálogo en Netbeans
`serviceName`, elemento de la anotación `@WebService`
servidor Apache Tomcat
servidor de aplicaciones
servidor GlassFish
SOAP (Protocolo simple de acceso a objetos)
Sun Java Studio Creator 2
Sun Java System Application Server
`Tester`, página Web de Sun Java System Application Server
tipo personalizado
transacciones B2B (negocio a negocio)
transacciones de negocio a negocio (B2B)
Transferencia representativa de estado (REST)
Web Application, proyecto en Netbeans
`WebServiceContext`, interfaz
WS-I Basic Profile 1.1 (BP 1.1)

Ejercicios de autoevaluación

28.1 Conteste con *verdadero* o *falso* a cada una de las siguientes proposiciones; en caso de ser *falso*, explique por qué.
a) Todos los métodos de una clase de servicio Web pueden ser invocados por los clientes de ese servicio Web.
b) Al consumir un servicio Web en una aplicación cliente creada en Netbeans, debemos crear la clase proxy que permita al cliente comunicarse con el servicio Web.
c) Una clase proxy que se comunica con un servicio Web generalmente usa SOAP para enviar y recibir mensajes.
d) El rastreo de sesiones está habilitado de manera automática en un cliente de un servicio Web.
e) Los métodos Web no se pueden declarar como `static`.

f) Un tipo definido por el usuario que se utilice en un servicio Web debe definir métodos *get* y *set* para cualquier propiedad que se vaya a serializar.

28.2 Complete los siguientes enunciados.

a) Cuando se envían mensajes entre una aplicación y un servicio Web mediante SOAP, cada mensaje se coloca en un(a) ________________.

b) Un servicio Web en java es un ________________; no necesita implementar ninguna interfaz o extender una clase.

c) Las peticiones a los servicios Web se transportan generalmente a través de Internet, mediante el protocolo ________________.

d) Para establecer el nombre expuesto de un método Web, utilice el elemento ________________ de la anotación `@WebMethod`.

e) La ________________ transforma a un objeto en un formato que se pueda enviar entre un servicio Web y un cliente.

Respuestas a los ejercicios de autoevaluación

28.1 a) Falso. Sólo los métodos declarados con la anotación `@WebMethod` pueden ser invocados por los clientes del servicio Web. b) Falso. Netbeans crea la clase proxy cuando agregamos un cliente de servicio Web a la aplicación. c) Verdadero. d) Falso. En el marco de trabajo JAX-WS 2.0, el cliente debe indicar si desea permitir que el servicio Web mantenga información sobre la sesión. Primero hay que convertir el objeto proxy al tipo de la interfaz `BindingProvider`, y después hay que usar el método `getRequestContext` de `BindingProvider` para obtener el objeto `RequestContext`. Por último, hay que usar el método `put` del objeto `RequestContext` para establecer la propiedad `BindingProvider.SESSION_MAINTAIN_PROPERTY` a `true`. e) Verdadero. f) Verdadero.

28.2 a) mensaje SOAP o envoltura SOAP.

b) POJO (Objeto Java simple).

c) HTTP.

d) `operationName`.

e) Serialización en XML.

Ejercicios

28.3 *(Servicio Web Libreta Telefónica)* Cree un servicio Web que almacene las entradas en una libreta telefónica, en la base de datos `LibretaDireccionesBD`, y una aplicación cliente Web que consuma este servicio. Use los pasos de la sección 28.7.1 para crear la base de datos `LibretaTelefonica`. Esta base sólo debe contener una tabla `(LibretaDirecciones)` con tres columnas: `ApellidoPaterno`, `PrimerNombre` y `NumeroTelefonico`, cada una de tipo `VARCHAR`. Las columnas `ApellidoPaterno` y `PrimerNombre` deben almacenar hasta 30 caracteres. La columna `NumeroTelefonico` debe soportar números telefónicos de la forma `(800) 555-1212`, que contienen 14 caracteres.

Dé al usuario cliente la capacidad de escribir un nuevo contacto (método Web `agregarEntrada`) y buscar contactos por apellido paterno (método Web `getEntradas`). Pase sólo objetos `String` como argumentos para el servicio Web. El método Web `getEntradas` debe devolver un arreglo de objetos `String` que contenga las entradas que coincidan en la libreta de direcciones. Cada objeto `String` en el arreglo debe consistir en el apellido paterno, primer nombre y número telefónico para una entrada en la libreta de direcciones. Estos valores deben separarse mediante comas.

La consulta SELECT para buscar una entrada en `LibretaDirecciones` por apellido paterno debería ser:

```
SELECT ApellidoPaterno, PrimerNombre, NumeroTelefonico
FROM LibretaDirecciones
WHERE (ApellidoPaterno = ApellidoPaterno)
```

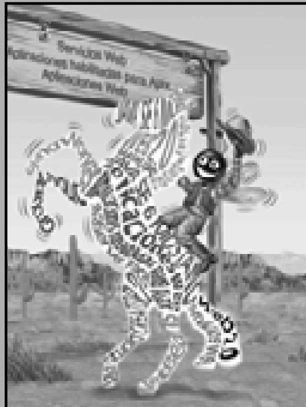
La instrucción `INSERT` para insertar una nueva entrada en la base de datos `LibretaDirecciones` debe ser:

```
INSERT INTO LibretaDirecciones (ApellidoPaterno, PrimerNombre, NumeroTelefonico)
VALUES (ApellidoPaterno, PrimerNombre, NumeroTelefonico)
```

28.4 *(Modificación al servicio Web Libreta telefónica)* Modifique el ejercicio 28.3, de manera que utilice una clase llamada `EntradaLibretaTelefonica` para representar a una fila en la base de datos. Al agregar contactos, la aplicación cliente debe proporcionar objetos de tipo `EntradaLibretaDirecciones` al servicio Web, y debe recibir objetos de tipo `EntradaLibretaDirecciones` a la hora de buscar contactos.

28.5 *(Modificación al servicio Web Blackjack)* Modifique el ejemplo del servicio Web Blackjack en la sección 28.6 para incluir la clase `Carta`. Modifique el método Web `repartirCarta`, de manera que devuelva un objeto de tipo `Carta`, y modifique el método Web `obtenerValorMano`, de manera que reciba un arreglo de objetos `Carta` del cliente. Modifique además la aplicación cliente para llevar la cuenta de qué cartas se han repartido, mediante el uso de objetos `ArrayList` de objetos `Carta`. La clase proxy creada por Netbean considerará el parámetro del arreglo de un método Web como un objeto `List`, de manera que podemos pasar estos objetos `ArrayList` de objetos `Carta` directamente al método `obtenerValorMano`. Su clase `Carta` deberá incluir métodos *set* y *get* para la cara y el palo de la carta.

29



Salida con formato

Todas las noticias que puede imprimir.

—Adolph S. Ochs

¿Qué loca persecución? ¿De qué aprieto escapar?

—John Keats

No elimines el punto de referencia en el límite de los campos.

—Amenehope

OBJETIVOS

En este capítulo aprenderá a:

- Trabajar con los flujos de entrada y salida.
- Usar el formato `printf`.
- Imprimir con anchuras de campo y precisiones.
- Usar banderas de formato en la cadena de formato de `printf`.
- Imprimir con un índice como argumento.
- Imprimir literales y secuencias de escape.
- Dar formato a la salida con la clase `Formatter`.

Plan general

29.1 Introducción

Una parte importante de la solución a cualquier problema es la presentación de los resultados. En este capítulo, hablaremos sobre las características de formato del método **`printf`** y la clase `Formatter` (paquete `java.util`). El método `printf` aplica formato a los datos y los imprime en el flujo de salida estándar (`System.out`). La clase **`Formatter`** aplica formato a los datos y los imprime en un destino especificado, como una cadena o un flujo de salida de archivo.

Anteriormente en este libro hablamos sobre muchas de las características de `printf`. En este capítulo se sintetizan esas características y se presentan otras, como mostrar datos de fecha y hora en varios formatos, reordenar la salida con base en el índice del argumento, y mostrar números y cadenas con varias banderas.

29.2 Flujos

Por lo general, las operaciones de entrada y salida se llevan a cabo con flujos, los cuales son secuencias de bytes. En las operaciones de entrada, los bytes fluyen de un dispositivo (como un teclado, una unidad de disco, una conexión de red) a la memoria principal. En las operaciones de salida, los bytes fluyen de la memoria principal a un dispositivo (como una pantalla, una impresora, una unidad de disco, una conexión de red).

Cuando empieza la ejecución de un programa, tres flujos se conectan a éste de manera automática. Por lo común, el flujo de entrada estándar se conecta al teclado, y el flujo de salida estándar se conecta a la pantalla. Un tercer flujo, el **flujo de error estándar** (`System.err`), se conecta generalmente a la pantalla y se utiliza para imprimir mensajes de error en ésta, de manera que puedan verse de inmediato; aún y cuando el flujo de salida estándar esté escribiendo en un archivo. Generalmente, los sistemas operativos permiten redirigir estos flujos a otros dispositivos. En el capítulo 14, Archivos y flujos, y en el capítulo 24, Redes, se describen los flujos con detalle.

29.3 Aplicación de formato a la salida con `printf`

Con `printf` podemos lograr un formato preciso en la salida. [*Nota:* Java SE 5 tomó prestada esta característica del lenguaje de programación C]. El método `printf` cuenta con las siguientes herramientas de formato, cada una de las cuales se verá en este capítulo:

1. Redondeo de valores de punto flotante a un número indicado de posiciones decimales.
2. Alineación de una columna de números con puntos decimales, que aparezcan uno encima de otro.
3. Justificación a la derecha y justificación a la izquierda de los resultados.

4. Inserción de caracteres literales en posiciones precisas de una línea de salida.
5. Representación de números de punto flotante en formato exponencial.
6. Representación de enteros en formato octal y hexadecimal (vea el apéndice E, Sistemas numéricos, para obtener más información acerca de los valores octales y hexadecimales).
7. Visualización de todo tipo de datos con anchuras de campo de tamaño fijo y precisiones.
8. Visualización de fechas y horas en diversos formatos.

Cada llamada a `printf` proporciona como primer argumento una **cadena de formato**, la cual describe el formato de salida. La cadena de formato puede consistir en **texto fijo** y **especificadores de formato**. El texto fijo se imprime mediante `printf` de igual forma que como se imprimiría mediante los métodos `print` o `println` de `System.out`. Cada especificador de formato es un receptáculo para un valor y especifica el tipo de datos a imprimir. Los especificadores de formato también pueden incluir información de formato opcional.

En su forma más simple, cada especificador de formato empieza con un signo de porcentaje (`%`) y va seguido de un **carácter de conversión** que representa el tipo de datos del valor a imprimir. Por ejemplo, el especificador de formato `%s` es un receptáculo para una cadena, y el especificador de formato `%d` es un receptáculo para un valor `int`. La información de formato opcional se especifica entre el signo de porcentaje y el carácter de conversión. La información de formato opcional incluye un índice como argumento, banderas, anchura de campo y precisión. A lo largo de este capítulo, definiremos cada uno de estos elementos, y mostraremos ejemplos.

29.4 Impresión de enteros

Un entero es un número completo, como `776`, `0` o `-52`, que no contiene punto decimal. Los valores enteros se muestran en uno de varios formatos. En la figura 29.1 se describen los **caracteres de conversión integrales**.

En la figura 29.2 se imprime un entero usando cada una de las conversiones integrales. En las líneas 9 a 10, observe que el signo positivo no se muestra de manera predeterminada, pero el signo negativo sí. Más adelante en este capítulo (figura 29.14) veremos cómo forzar a que se impriman los signos positivos.

El método `printf` tiene la forma

`printf(` *cadena-de-formato*`,` *lista-de-argumentos* `);`

en donde *cadena-de-formato* describe el formato de salida, y *lista-de-argumentos* contiene los valores que corresponden a cada especificador de formato en *cadena-de-formato*. Puede haber muchos especificadores de formato en una cadena de formato.

Cada cadena de formato en las líneas 8 a 10 especifica que `printf` debe imprimir un entero decimal (`%d`) seguido de un carácter de nueva línea. En la posición del especificador de formato, `printf` sustituye el valor del primer argumento después de la cadena de formato. Si la cadena de formato contiene varios especificadores de formato, en cada posición del siguiente especificador de formato, `printf` sustituirá el valor del siguiente argumento en la lista de argumentos. El especificador de formato `%o` en la línea 11 imprime el entero en formato octal. El especificador de formato `%x` en la línea 12 imprime el entero en formato hexadecimal. El especificador de formato `%X` en la línea 13 imprime el entero en formato hexadecimal, con letras mayúsculas.

Carácter de conversión	Descripción
`d`	Muestra un entero decimal (base 10).
`o`	Muestra un entero octal (base 8).
`x` o `X`	Muestra un entero hexadecimal (base 16). `X` hace que se muestren los dígitos del 0 al 9 y la letras `A` a la `F`, y `x` hace que se muestren los dígitos del 0 al 9 y las letras de la a a la f.

Figura 29.1 | Caracteres de conversión de enteros.

```
// Fig. 29.2: PruebaConversionEnteros.java
// Uso de los caracteres de conversión integrales.

public class PruebaConversionEnteros
{
   public static void main( String args[] )
   {
      System.out.printf( "%d\n", 26 );
      System.out.printf( "%d\n", +26 );
      System.out.printf( "%d\n", -26 );
      System.out.printf( "%o\n", 26 );
      System.out.printf( "%x\n", 26 );
      System.out.printf( "%X\n", 26 );
   } // fin de main
} // fin de la clase PruebaConversionEnteros
```

```
26
26
-26
32
1a
1A
```

Figura 29.2 | Uso de caracteres de conversión de enteros.

29.5 Impresión de números de punto flotante

Un valor de punto flotante contiene un punto decimal, como en `33.5`, `0.0` o `-657.983`. Los valores de punto flotante se muestran en uno de varios formatos. En la figura 29.3 se describen las conversiones de punto flotante. Los **caracteres de conversión `e` y `E`** muestran valores de punto flotante en **notación científica computarizada** (también conocida como **notación exponencial**). La notación exponencial es el equivalente computacional de la notación científica que se utiliza en las matemáticas. Por ejemplo, el valor `150.4582` se representa en notación científica matemática de la siguiente manera:

$$1.504582 \times 10^{2}$$

y se representa en notación exponencial como

```
1.504582e+02
```

en Java. Esta notación indica que `1.504582` se multiplica por `10` elevado a la segunda potencia (`e+02`). La **`e`** representa al "exponente".

Carácter de conversión	Descripción
`e` o `E`	Muestra un valor de punto flotante en notación exponencial. Cuando se utiliza el carácter de conversión `E`, la salida se muestra en letras mayúsculas.
`f`	Muestra un valor de punto flotante en formato decimal.
`g` o `G`	Muestra un valor de punto flotante en el formato de punto flotante `f` o en el formato exponencial `e`, con base en la magnitud del valor. Si la magnitud es menor que 10^{-3}, o si es mayor o igual que 10^{7}, el valor de punto flotante se imprime con `e` (o `E`). En cualquier otro caso, el valor se imprime en el formato `f`. Cuando se utiliza el carácter de conversión `G`, la salida se muestra en letras mayúsculas.
`a` o `A`	Muestra un número de punto flotante en formato hexadecimal. Cuando se usa el carácter de conversión `A`, la salida se muestra en letras mayúsculas.

Figura 29.3 | Caracteres de conversión de punto flotante.

Los valores que se imprimen con los caracteres de conversión `e`, `E` y `f` se muestran con seis dígitos de precisión en el lado derecho del punto decimal de manera predeterminada (por ejemplo, `1.045921`); otras precisiones se deben especificar de manera explícita. Para los valores impresos con el carácter de conversión `g`, la precisión representa el número total de dígitos mostrados, excluyendo el exponente. El valor predeterminado es de seis dígitos (por ejemplo, `12345678.9` se muestra como `1.23457e+07`). El **carácter de conversión f** siempre imprime por lo menos un dígito a la izquierda del punto decimal. Los caracteres de conversión `e` y `E` imprimen una `e` minúscula y una `E` mayúscula antes del exponente, y siempre imprimen sólo un dígito a la izquierda del punto decimal. El redondeo ocurre si el valor al que se está dando formato tiene más dígitos significativos que la precisión.

El **carácter de conversión g** (o `G`) imprime en formato `e` (`E`) o `f`, dependiendo del valor de punto flotante. Por ejemplo, los valores `0.0000875`, `87500000.0`, `8.75`, `87.50` y `875.0` se imprimen como `8.750000e-05`, `8.750000e+07`, `8.750000`, `87.500000` y `875.000000` con el carácter de conversión `g`. El valor `0.0000875` utiliza la notación `e` ya que la magnitud es menor que 10^{-3}. El valor `87500000.0` utiliza la notación `e`, debido a que la magnitud es mayor que 10^{7}. En la figura 29.4 se muestra cada uno de los caracteres de conversión de punto flotante.

```java
// Fig. 29.4: PruebaPuntoFlotante.java
// Uso de los caracteres de conversión de punto flotante.

public class PruebaPuntoFlotante
{
   public static void main( String args[] )
   {
      System.out.printf( "%e\n", 12345678.9 );
      System.out.printf( "%e\n", +12345678.9 );
      System.out.printf( "%e\n", -12345678.9 );
      System.out.printf( "%E\n", 12345678.9 );
      System.out.printf( "%f\n", 12345678.9 );
      System.out.printf( "%g\n", 12345678.9 );
      System.out.printf( "%G\n", 12345678.9 );
   } // fin de main
} // fin de la clase PruebaPuntoFlotante
```

```
1.234568e+07
1.234568e+07
-1.234568e+07
1.234568E+07
12345678.900000
1.23457e+07
1.23457E+07
```

Figura 29.4 | Uso de los caracteres de conversión de punto flotante.

29.6 Impresión de cadenas y caracteres

Los caracteres de conversión `c` y `s` se utilizan para imprimir caracteres individuales y cadenas, respectivamente. El carácter de conversión `s` también puede imprimir objetos con los resultados de las llamadas implícitas al método `toString`. Los **caracteres de conversión c** y `C` requieren un argumento `char`. Los **caracteres de conversión s** y `S` pueden recibir un objeto `String` o cualquier objeto `Object` (se incluyen todas las subclases de `Object`) como argumento. Cuando se pasa un objeto al carácter de conversión `s`, el programa utiliza de manera implícita el método `toString` del objeto para obtener la representación `String` del objeto. Cuando se utilizan los caracteres de conversión `C` y `S`, la salida se muestra en letras mayúsculas. El programa que se muestra en la figura 29.5 imprime en pantalla caracteres, cadenas y objetos con los caracteres de conversión `c` y `s`. Observe que se realiza una conversión autoboxing en la línea 10, cuando se asigna una constante `int` a un objeto `Integer`. En la línea 15 se asocia un objeto `Integer` como argumento para el carácter de conversión `s`, el cual invoca de manera implícita al método `toString` para obtener el valor entero. Observe que también puede imprimir en pantalla un objeto `Integer` mediante el uso del especificador de formato `%d`. En este caso, se realizará una conversión unboxing con el valor `int` en el objeto `Integer` y se imprimirá en pantalla.

```java
// Fig. 29.5: ConversionCadenasChar.java
// Uso de los caracteres de conversión de cadenas y caracteres.

public class ConversionCadenasChar
{
   public static void main( String args[] )
   {
      char caracter= 'A';  // inicializa el char
      String cadena = "Esta tambien es una cadena"; // objeto String
      Integer entero = 1234;  // inicializa el entero (autoboxing)

      System.out.printf( "%c\n", caracter );
      System.out.printf( "%s\n", "Esta es una cadena" );
      System.out.printf( "%s\n", cadena );
      System.out.printf( "%S\n", cadena );
      System.out.printf( "%s\n", entero ); // llamada implícita a toString
   } // fin de main
} // fin de la clase ConversionCadenasChar
```

```
A
Esta es una cadena
Esta tambien es una cadena
ESTA TAMBIEN ES UNA CADENA
1234
```

Figura 29.5 | Uso de los caracteres de conversión de cadenas y caracteres.



Error común de programación 29.1

El uso de `%c` *para imprimir una cadena produce una excepción* `IllegalFormatConversionException`*; una cadena no se puede convertir en un carácter.*

29.7 Impresión de fechas y horas

Con el **carácter de conversión `t` o `T`**, podemos imprimir fechas y horas en diversos formatos. El carácter de conversión `t` o `T` siempre va seguido de un **carácter de sufijo de conversión** que especifica el formato de fecha y/o de hora. Cuando se utiliza el carácter de conversión `T`, la salida se muestra en letras mayúsculas. En la figura 29.6 se listan los caracteres de sufijo de conversión comunes para aplicar formato a las **composiciones de fecha y hora** que muestran tanto la fecha como la hora. En la figura 29.7 se listan los caracteres de sufijo de conversión comunes para aplicar formato a las fechas. En la figura 29.8 se listan los caracteres de sufijo de conversión comunes para aplicar formato a las horas. Para ver la lista completa de caracteres de sufijo de conversión, visite el sitio Web `java.sun.com/javase/6/docs/api/java/util/Formatter.html`.

Carácter de sufijo de conversión	Descripción
`c`	Muestra la fecha y hora con el formato `dia mes fecha hora:minuto:segundo zona-horaria año` con tres caracteres para `dia` y `mes`, dos dígitos para `fecha`, `hora`, `minuto` y `segundo`, y cuatro dígitos para `año`; por ejemplo, `Mié Mar 03 16:30:25 GMT -05:00 2004`. Se utiliza el reloj de 24 horas. En este ejemplo, `GMT -05:00` es la zona horaria.
`F`	Muestra la fecha con el formato `año-mes-dia` con cuatro dígitos para el `año` y dos dígitos para el `mes` y la `fecha` (por ejemplo, `2004-05-04`).

Figura 29.6 | Caracteres de sufijo de conversión de composiciones de fecha y hora. (Parte 1 de 2).

Carácter de sufijo de conversión	Descripción
`D`	Muestra la fecha con el formato `mes/dia/año`, con dos dígitos para el `mes`, `dia` y `año` (por ejemplo, `03/03/04`).
`r`	Muestra la hora con el formato `hora:minuto:segundo AM|PM`, con dos dígitos para la `hora`, `minuto` y `segundo` (por ejemplo, `04:30:25 PM`). Se utiliza el reloj de 12 horas.
`R`	Muestra la hora con el formato `hora:minuto`, con dos dígitos para la `hora` y el `minuto` (por ejemplo, `16:30`). Se utiliza el reloj de 24 horas.
`T`	Muestra la hora con el formato `hora:minuto:segundo`, con dos dígitos para la `hora`, `minuto` y `segundo` (por ejemplo, `16:30:25`). Se utiliza el reloj de 24 horas.

Figura 29.6 | Caracteres de sufijo de conversión de composiciones de fecha y hora. (Parte 2 de 2).

Carácter de sufijo de conversión	Descripción
`A`	Muestra el nombre completo del día de la semana (por ejemplo, `Miércoles`).
`a`	Muestra el nombre corto de tres caracteres del día de la semana (por ejemplo, `Mié`).
`B`	Muestra el nombre completo del mes (por ejemplo, `Marzo`).
`b`	Muestra el nombre corto de tres caracteres del mes (por ejemplo, `Mar`).
`d`	Muestra el día del mes con dos dígitos, rellenando con ceros a la izquierda si es necesario (por ejemplo, `03`).
`m`	Muestra el mes con dos dígitos, rellenando con ceros a la izquierda si es necesario (por ejemplo, `07`).
`e`	Muestra el día del mes sin ceros a la izquierda (por ejemplo, `3`).
`Y`	Muestra el año con cuatro dígitos (por ejemplo, `2004`).
`y`	Muestra los dos últimos dígitos del año con ceros a la izquierda, según sea necesario (por ejemplo, `04`).
`j`	Muestra el día del año con tres dígitos, rellenando con ceros a la izquierda según sea necesario (por ejemplo, `016`).

Figura 29.7 | Caracteres de sufijo de conversión para aplicar formato a las fechas.

Carácter de sufijo de conversión	Descripción
`H`	Muestra la hora en el reloj de 24 horas, con un cero a la izquierda si es necesario (por ejemplo, `16`).
`I`	Muestra la hora en el reloj de 12 horas, con un cero a la izquierda si es necesario (por ejemplo, `04`).
`k`	Muestra la hora en el reloj de 24 horas sin ceros a la izquierda (por ejemplo, `16`).
`l`	Muestra la hora en el reloj de 12 horas sin ceros a la izquierda (por ejemplo, `4`).

Figura 29.8 | Caracteres de sufijo de conversión para aplicar formato a las horas. (Parte 1 de 2).

Carácter de sufijo de conversión	Descripción
M	Muestra los minutos con un cero a la izquierda, si es necesario (por ejemplo, 06).
S	Muestra los segundos con un cero a la izquierda, si es necesario (por ejemplo, 05).
Z	Muestra la abreviación para la zona horaria (por ejemplo, GMT -05:00, que representa a la Hora estándar occidental, la cual se encuentra a 5 horas de retraso de la Hora del Meridiano de Greenwich).
p	Muestra las iniciales de mañana o tarde en minúsculas (por ejemplo, pm).
P	Muestra el marcador de mañana o tarde en mayúsculas (por ejemplo, PM).

Figura 29.8 | Caracteres de sufijo de conversión para aplicar formato a las horas. (Parte 2 de 2).

En la figura 29.9 se utiliza el carácter de conversión `t` con los caracteres de sufijo de conversión para mostrar fechas y horas en diversos formatos. El carácter de conversión `t` requiere que su correspondiente argumento sea de tipo `long`, `Long`, **`Calendar`** o **`Date`** (ambas clases se encuentran en el paquete `java.util`); los objetos de cada una de estas clases pueden representar fechas y horas. La clase `Calendar` es la preferida para este propósito, ya que ciertos constructores y métodos en la clase `Date` se sustituyen por los de la clase `Calendar`. En la línea 10 se invoca el método `static` **`getInstance`** de `Calendar` para obtener un calendario con la fecha y hora actuales. En las líneas 13 a 17, 20 a 22 y 25 a 26 se utiliza este objeto `Calendar` en instrucciones `printf` como el valor al que se aplicará formato con el carácter de conversión `t`. Observe que en las líneas 20 a 22 y 25 a 26 se utiliza el **índice como argumento** opcional (`"1$"`) para indicar que todos los especificadores de formato en la cadena de formato utilizan el primer argumento después de la cadena de formato en la lista de argumentos. En la sección 29.11 aprenderá más acerca de los índices como argumentos. Al usar el índice como argumento, se elimina la necesidad de listar repetidas veces el mismo argumento.

```
1  // Fig. 29.9: PruebaFechaHora.java
2  // Aplicación de formato a fechas y horas con los caracteres de conversión t y T.
3  import java.util.Calendar;
4
5  public class PruebaFechaHora
6  {
7     public static void main( String args[] )
8     {
9        // obtiene la fecha y hora actuales
10       Calendar fechaHora = Calendar.getInstance();
11
12       // impresión con caracteres de conversión para composiciones de fecha/hora
13       System.out.printf( "%tc\n", fechaHora );
14       System.out.printf( "%tF\n", fechaHora );
15       System.out.printf( "%tD\n", fechaHora );
16       System.out.printf( "%tr\n", fechaHora );
17       System.out.printf( "%tT\n", fechaHora );
18
19       // impresión con caracteres de conversión para fechas
20       System.out.printf( "%1$tA, %1$tB %1$td, %1$tY\n", fechaHora );
21       System.out.printf( "%1$TA, %1$TB %1$Td, %1$TY\n", fechaHora );
22       System.out.printf( "%1$ta, %1$tb %1$te, %1$ty\n", fechaHora );
23
24       // impresión con caracteres de conversión para horas
25       System.out.printf( "%1$tH:%1$tM:%1$tS\n", fechaHora );
```

Figura 29.9 | Aplicación de formato a fechas y horas con los caracteres de conversión `t`. (Parte 1 de 2).

```
        System.out.printf( "%1$tZ %1$tI:%1$tM:%1$tS %Tp", fechaHora );
    } // fin de main
} // fin de la clase PruebaFechaHora
```

```
mié nov 07 11:54:30 CST 2007
2007-11-07
11/07/07
11:54:30 AM
11:54:30
miércoles, noviembre 07, 2007
MIÉRCOLES, NOVIEMBRE 07, 2007
mié, nov 7, 07
11:54:30
CST 11:54:30 AM
```

Figura 29.9 | Aplicación de formato a fechas y horas con los caracteres de conversión `t`. (Parte 2 de 2).

29.8 Otros caracteres de conversión

El resto de los caracteres de conversión son `b`, `B`, `h`, `H`, `%` y `n`. Éstos se describen en la figura 29.10.

En las líneas 9 y 10 de la figura 29.11 se utiliza `%b` para imprimir el valor de los valores booleanos `false` y `true`. En la línea 11 se asocia un objeto `String` a `%b`, el cual devuelve `true` debido a que no es `null`. En la línea 12 se asocia un objeto `null` a `%B`, el cual muestra `FALSE` ya que `prueba` es `null`. En las líneas 13 y 14 se utiliza `%h` para imprimir las representaciones de cadena de los valores de código hash para las cadenas `"hola"` y `"Hola"`. Estos valores se podrían utilizar para almacenar o colocar las cadenas en un objeto `Hashtable` o `HashMap` (los cuales vimos en el capítulo 19, Colecciones). Observe que los valores de código hash para estas dos cadenas difieren, ya que una cadena empieza con letra minúscula y la otra con letra mayúscula. En la línea 15 se utiliza `%H` para imprimir `null` en letras mayúsculas. Las últimas dos instrucciones `printf` (líneas 16 y 17) utilizan `%%` para imprimir el carácter `%` en una cadena, y `%n` para imprimir un separador de línea específico de la plataforma.



Error común de programación 29.2

Tratar de imprimir un carácter de porcentaje literal mediante el uso de `%` en vez de `%%` en la cadena de formato podría provocar un error lógico difícil de detectar. Cuando aparece el `%` en una cadena de formato, debe ir seguido de un carácter de conversión en la cadena. El signo de por ciento individual podría ir seguido accidentalmente de un carácter de conversión legítimo, con lo cual se produciría un error lógico.

Carácter de conversión	Descripción
`b` o `B`	Imprime `"true"` o `"false"` para el valor de un `boolean` o `Boolean`. Estos caracteres de conversión también pueden aplicar formato al valor de cualquier referencia. Si la referencia no es `null`, se imprime `"true"`; en caso contrario, se imprime `"false"`. Cuando se utiliza el carácter de conversión `B`, la salida se muestra en letras mayúsculas.
`h` o `H`	Imprime la representación de cadena del valor de código hash de un objeto en formato hexadecimal. Si el correspondiente argumento es `null`, se imprime `"null"`. Cuando se utiliza el carácter de conversión `H`, la salida se muestra en letras mayúsculas.
`%`	Imprime el carácter de por ciento.
`n`	Imprime el separador de línea específico de la plataforma (por ejemplo, `\r\n` en Windows o `\n` en UNIX/LINUX).

Figura 29.10 | Otros especificadores de conversión.

```
// Fig. 29.11: OtrasConversiones.java
// Uso de los caracteres de conversión b, B, h, H, % y n.

public class OtrasConversiones
{
   public static void main( String args[] )
   {
      Object prueba = null;
      System.out.printf( "%b\n", false );
      System.out.printf( "%b\n", true );
      System.out.printf( "%b\n", "Prueba" );
      System.out.printf( "%B\n", prueba );
      System.out.printf( "El codigo hash de \"hola\" es %h\n", "hola" );
      System.out.printf( "El codigo hash de \"Hola\" es %h\n", "Hola" );
      System.out.printf( "El codigo hash de null es %H\n", prueba );
      System.out.printf( "Impresion de un %% en una cadena de formato\n" );
      System.out.printf( "Impresion de una nueva linea %nla siguiente linea empieza
      aqui" );
   } // fin de main
} // fin de la clase OtrasConversiones
```

```
false
true
true
FALSE
El codigo hash de "hola" es 30f4bc
El codigo hash de "Hola" es 2268dc
El codigo hash de null es NULL
Impresion de un % en una cadena de formato
Impresion de una nueva linea
la siguiente linea empieza aquí
```

Figura 29.11 | Uso de los caracteres de conversión `b`, `B`, `h`, `H`, `%` y `n`.

29.9 Impresión con anchuras de campo y precisiones

El tamaño exacto de un campo en el que se imprimen datos se especifica mediante una **anchura de campo**. Si la anchura de campo es mayor que los datos que se van a imprimir, éstos se justificarán a la derecha dentro de ese campo, de manera predeterminada. (En la sección 29.10 demostraremos la justificación a la izquierda). El programador inserta un entero que representa la anchura de campo entre el signo de porcentaje (%) y el carácter de conversión (por ejemplo, `%4d`) en el especificador de formato. En la figura 29.12 se imprimen dos grupos de cinco números cada uno, y se justifican a la derecha los números que contienen menos dígitos que la anchura de campo. Observe que la anchura de campo se incrementa para imprimir valores más anchos que el campo, y que el signo menos para un valor negativo utiliza una posición de carácter en el campo. Además, si no se especifica la anchura del campo, los datos se imprimen en todas las posiciones que sean necesarias. Las anchuras de campo pueden utilizarse con todos los especificadores de formato, excepto el separador de línea (`%n`).



Error común de programación 29.3

Si no se proporciona una anchura de campo suficientemente extensa como para manejar un valor a imprimir, se pueden desplazar los demás datos que se impriman, con lo que se producirán resultados confusos. ¡Debe conocer sus datos!

El método `printf` también proporciona la habilidad de especificar la precisión con la que se van a imprimir los datos. La precisión tiene distintos significados para los diferentes tipos. Cuando se utiliza con los caracteres de conversión de punto flotante `e` y `f`, la precisión es el número de dígitos que aparecen después del punto decimal.

```
// Fig. 29.12: PruebaAnchuraCampo.java
// Justificación a la derecha de enteros en campos.

public class PruebaAnchuraCampo
{
   public static void main( String args[] )
   {
      System.out.printf( "%4d\n", 1 );
      System.out.printf( "%4d\n", 12 );
      System.out.printf( "%4d\n", 123 );
      System.out.printf( "%4d\n", 1234 );
      System.out.printf( "%4d\n\n", 12345 ); // datos demasiado extensos

      System.out.printf( "%4d\n", -1 );
      System.out.printf( "%4d\n", -12 );
      System.out.printf( "%4d\n", -123 );
      System.out.printf( "%4d\n", -1234 ); // datos demasiado extensos
      System.out.printf( "%4d\n", -12345 ); // datos demasiado extensos
   } // fin de main
} // fin de la clase PruebaAnchuraCampo
```

```
   1
  12
 123
1234
12345

  -1
 -12
-123
-1234
-12345
```

Figura 29.12 | Justificación a la derecha de enteros en campos.

Cuando se utiliza con el carácter de conversión `g`, la precisión es el número máximo de dígitos significativos a imprimir. Cuando se utiliza con el carácter de conversión `s`, la precisión es el número máximo de caracteres a escribir de la cadena. Para utilizar la precisión, se debe colocar entre el signo de porcentaje y el especificador de conversión un punto decimal (`.`), seguido de un entero que representa la precisión. En la figura 29.13 se muestra el uso de la precisión en las cadenas de formato. Observe que, cuando se imprime un valor de punto flotante con una precisión menor que el número original de posiciones decimales en el valor, éste se redondea. Además, observe que el especificador de formato `%.3g` indica que el número total de dígitos utilizados para mostrar el valor de punto flotante es `3`. Como el valor tiene tres dígitos a la izquierda del punto decimal, se redondea a la posición de las unidades.

La anchura de campo y la precisión pueden combinarse, para lo cual se coloca la anchura de campo, seguida de un punto decimal, seguido de una precisión entre el signo de porcentaje y el carácter de conversión, como en la siguiente instrucción:

```
printf( "%9.3f", 123.456789 );
```

la cual muestra `123.457` con tres dígitos a la derecha del punto decimal, y se justifica a la derecha en un campo de nueve dígitos; antes del número se colocarán dos espacios en blanco en su campo.

29.10 Uso de banderas en la cadena de formato de `printf`

Pueden usarse varias banderas con el método `printf` para suplementar sus herramientas de formato de salida. Hay siete banderas disponibles para usarlas en las cadenas de formato (figura 29.14).

```
// Fig 29.13: PruebaPrecision.java
// Uso de la precisión para números de punto flotante y cadenas.
public class PruebaPrecision
{
   public static void main( String args[] )
   {
      double f = 123.94536;
      String s = "Feliz Cumpleanios";

      System.out.printf( "Uso de la precision para numeros de punto flotante\n" );
      System.out.printf( "\t%.3f\n\t%.3e\n\t%.3g\n\n", f, f, f );

      System.out.printf( "Uso de la precision para las cadenas\n" );
      System.out.printf( "\t%.11s\n", s );
   } // fin de main
} // fin de la clase PruebaPrecision
```

```
Uso de la precision para numeros de punto flotante
        123.945
        1.239e+02
        124

Uso de la precision para las cadenas
        Feliz Cumpl
```

Figura 29.13 | Uso de la precisión para los números de punto flotante y las cadenas.

Bandera	Descripción
- (signo negativo)	Justifica a la izquierda la salida dentro del campo especificado.
+ (signo positivo)	Muestra un signo positivo antes de los valores positivos, y un signo negativo antes de los valores negativos.
espacio	Imprime un espacio antes de un valor positivo que no se imprime con la bandera +.
#	Antepone un 0 al valor de salida cuando se utiliza con el carácter de conversión octal o. Antepone 0x al valor de salida cuando se usa con el carácter de conversión hexadecimal x.
0 (cero)	Rellena un campo con ceros a la izquierda.
, (coma)	Usa el separador de miles específico para la configuración regional (es decir, ',' para los EUA), para mostrar números decimales y de punto flotante.
(	Encierra los números negativos entre paréntesis.

Figura 29.14 | Banderas de la cadena de formato.

Para usar una bandera en una cadena de formato, coloque la bandera justo a la derecha del signo de porcentaje. Pueden usarse varias banderas en el mismo especificador de formato. En la figura 29.15 se muestra la justificación a la derecha y la justificación a la izquierda de una cadena, un entero, un carácter y un número de punto flotante. Observe que la línea 9 sirve como mecanismo de conteo para la salida en la pantalla.

En la figura 29.16 se imprime un número positivo y un número negativo, cada uno con y sin la **bandera +**. Observe que el signo negativo se muestra en ambos casos, pero el signo positivo se muestra sólo cuando se utiliza la bandera +.

```
// Fig. 29.15: PruebaBanderaMenos.java
// Justificación a la derecha y justificación a la izquierda de los valores

public class PruebaBanderaMenos
{
   public static void main( String args[] )
   {
      System.out.println( "Columnas:" );
      System.out.println( "0123456789012345678901234567890123456789\n" );
      System.out.printf( "%10s%10d%10c%10f\n\n", "hello", 7, 'a', 1.23 );
      System.out.printf(
         "%-10s%-10d%-10c%-10f\n", "hola", 7, 'a', 1.23 );
   } // fin de main
} // fin de la clase PruebaBanderaMenos
```

```
Columnas:
0123456789012345678901234567890123456789

      hola         7         a  1.230000

hola      7         a         1.230000
```

Figura 29.15 | Justificación a la derecha y justificación a la izquierda de los valores.

En la figura 29.17 se antepone un espacio al número positivo mediante la **bandera de espacio**. Esto es útil para alinear números positivos y negativos con el mismo número de dígitos. Observe que el valor -547 no va precedido por un espacio en la salida, debido a su signo negativo. En la figura 29.18 se utiliza la **bandera #** para anteponer un 0 al valor octal, y **0x** para el valor hexadecimal.

En la figura 29.19 se combinan la bandera +, la **bandera 0** y la bandera de espacio para imprimir 452 en un campo con una anchura de 9, con un signo **+** y ceros a la izquierda; después se imprime 452 en un campo de anchura 9, usando sólo la bandera 0, y después se imprime 452 en un campo de anchura 9, usando sólo la bandera de espacio.

```
// Fig. 29.16: PruebaBanderaMas.java
// Impresión de números con y sin la bandera +.

public class PruebaBanderaMas
{
   public static void main( String args[] )
   {
      System.out.printf( "%d\t%d\n", 786, -786 );
      System.out.printf( "%+d\t%+d\n", 786, -786 );
   } // fin de main
} // fin de la clase PruebaBanderaMas
```

```
786     -786
+786    -786
```

Figura 29.16 | Impresión de números con y sin la bandera +.

En la figura 29.20 se utiliza la bandera de coma (,) para mostrar un número decimal y un número de punto flotante con el separador de miles. En la figura 29.21 se encierran números negativos entre paréntesis, usando la bandera (. Observe que el valor 50 no se encierra entre paréntesis en la salida, ya que es un número positivo.

```java
// Fig. 29.17: PruebaBanderaEspacio.java
// Impresión de un espacio antes de valores no negativos.

public class PruebaBanderaEspacio
{
   public static void main( String args[] )
   {
      System.out.printf( "% d\n% d\n", 547, -547 );
   } // fin de main
} // fin de la clase PruebaBanderaEspacio
```

```
 547
-547
```

Figura 29.17 | Uso de la bandera de espacio para imprimir un espacio antes de los valores no negativos.

```java
// Fig. 29.18: PruebaBanderaLibras.java
// Uso de la bandera # con los caracteres de conversión o y x.

public class PruebaBanderaLibras
{
   public static void main( String args[] )
   {
      int c = 31;       // inicializa c

      System.out.printf( "%#o\n", c );
      System.out.printf( "%#x\n", c );
   } // fin de main
} // fin de la clase PruebaBanderaLibras
```

```
037
0x1f
```

Figura 29.18 | Uso de la bandera # con los caracteres de conversión o y x.

```java
// Fig. 29.19: PruebaBanderaCero.java
// Impresión con la bandera 0 (cero) para rellenar con ceros a la izquierda.

public class PruebaBanderaCero
{
   public static void main( String args[] )
   {
      System.out.printf( "%+09d\n", 452 );
      System.out.printf( "%09d\n", 452 );
      System.out.printf( "% 9d\n", 452 );
   } // fin de main
} // fin de la clase PruebaBanderaCero
```

```
+00000452
000000452
      452
```

Figura 29.19 | Impresión con la bandera 0 (cero) para rellenar con ceros a la izquierda.

```
// Fig. 29.20: PruebaBanderaComa.java
// Uso de la bandera de coma (,) para mostrar números con el separador de miles.

public class PruebaBanderaComa
{
   public static void main( String args[] )
   {
      System.out.printf( "%,d\n", 58625 );
      System.out.printf( "%,.2f\n", 58625.21 );
      System.out.printf( "%,.2f", 12345678.9 );
   } // fin de main
} // fin de la clase PruebaBanderaComa
```

```
58,625
58,625.21
12,345,678.90
```

Figura 29.20 | Uso de la bandera de coma (,) para mostrar números con el separador de miles.

```
// Fig. 29.21: PruebaBanderaParentesis.java
// Uso de la bandera ( para colocar paréntesis alrededor de números negativos.

public class PruebaBanderaParentesis
{
   public static void main( String args[] )
   {
      System.out.printf( "%(d\n", 50 );
      System.out.printf( "%(d\n", -50 );
      System.out.printf( "%(.1e\n", -50.0 );
   } // fin de main
} // fin de la clase PruebaBanderaParentesis
```

```
50
(50)
(5.0e+01)
```

Figura 29.21 | Uso de la bandera (para colocar paréntesis alrededor de números negativos.

29.11 Impresión con índices como argumentos

Un **índice como argumento** es un entero opcional, seguido de un signo de $, el cual indica la posición del argumento en la lista de argumentos. Por ejemplo, en las líneas 20 a 21 y 24 a 25 de la figura 29.9 se utiliza el índice como argumento "1$" para indicar que todos los especificadores de formato utilizan el primer argumento en la lista de argumentos. Los índices como argumentos permiten a los programadores reordenar la salida, de manera que los argumentos en la lista de argumentos no necesariamente se encuentren en el orden de sus especificadores de formato correspondientes. Los índices como argumentos también ayudan a evitar argumentos duplicados. En la figura 29.22 se muestra cómo imprimir argumentos en la lista de argumentos en orden inverso, mediante el uso del índice como argumento.

```
// Fig. 29.22: PruebaIndiceArgumento
// Reordenamiento de la salida con los índices como argumentos.

public class PruebaIndiceArgumento
```

Figura 29.22 | Reordenamiento de la salida con los índices como argumentos. (Parte 1 de 2).

```
{
   public static void main( String args[] )
   {
      System.out.printf(
         "Lista de parametros sin reordenar: %s %s %s %s\n",
         "primero", "segundo", "tercero", "cuarto" );
      System.out.printf(
         "Lista de parametros despues de reordenar: %4$s %3$s %2$s %1$s\n",
         "primero", "segundo", "tercero", "cuarto" );
   } // fin de main
} // fin de la clase PruebaIndiceArgumento
```

```
Lista de parametros sin reordenar: primero segundo tercero cuarto
Lista de parametros despues de reordenar: cuarto tercero segundo primero
```

Figura 29.22 | Reordenamiento de la salida con los índices como argumentos. (Parte 2 de 2).

29.12 Impresión de literales y secuencias de escape

La mayoría de los caracteres literales que se imprimen en una instrucción `printf` sólo necesitan incluirse en la cadena de formato. Sin embargo, hay varios caracteres "problemáticos", como el signo de comillas dobles (") que delimita a la cadena de formato en sí. Varios caracteres de control, como el carácter de nueva línea y el de tabulación, deben representarse mediante secuencias de escape. Una secuencia de escape se representa mediante una barra diagonal inversa (\), seguida de un carácter de escape. En la figura 29.23 se listan las secuencias de escape y las acciones que producen.



Error común de programación 29.4

Tratar de imprimir un carácter de comillas dobles o un carácter de barra diagonal inversa como datos literales en una instrucción `printf`*, sin anteponer al carácter una barra diagonal inversa para formar una secuencia de escape apropiada, podría producir un error de sintaxis.*

Secuencia de escape	Descripción
`\'` (comilla sencilla)	Imprime el carácter de comilla sencilla (').
`\"` (doble comilla)	Imprime el carácter de doble comilla (").
`\\` (barra diagonal inversa)	Imprime el carácter de barra diagonal inversa (\).
`\b` (retroceso)	Desplaza el cursor una posición hacia atrás en la línea actual.
`\f` (nueva página o avance de página)	Desplaza el cursor al principio de la siguiente página lógica.
`\n` (nueva línea)	Desplaza el cursor al principio de la siguiente línea.
`\r` (retorno de carro)	Desplaza el cursor al principio de la línea actual.
`\t` (tabulador horizontal)	Desplaza el cursor hacia la siguiente posición del tabulador horizontal.

Figura 29.23 | Secuencias de escape.

29.13 Aplicación de formato a la salida con la clase `Formatter`

Hasta ahora, hemos visto cómo mostrar salida con formato en el flujo de salida estándar. ¿Qué deberíamos hacer si quisiéramos enviar salidas con formato a otros flujos de entrada o dispositivos, como un objeto `JTextArea` o un archivo? La solución recae en la clase `Formatter` (en el paquete `java.util`), la cual proporciona las mismas herramientas de formato que `printf`. `Formatter` es una clase utilitaria que permite a los programadores impri-

mir datos con formato hacia un destino especificado, como un archivo en el disco. De manera predeterminada, un objeto `Formatter` crea una cadena en la memoria. En la figura 29.24 se muestra cómo usar un objeto `Formatter` para crear una cadena con formato, la cual después se muestra en un cuadro de diálogo de mensaje.
En la línea 11 se crea un objeto `Formatter` mediante el uso del constructor predeterminado, por lo que este objeto creará una cadena en la memoria. Se incluyen otros constructores para que el programador pueda especificar el destino hacia el que se deben enviar los datos con formato. Para obtener más información, consulte la página `java.sun.com/javase/6/docs/api/java/util/Formatter.html`

En la línea 12 se invoca el método **`format`** para dar formato a la salida. Al igual que `printf`, el método `format` recibe una cadena de formato y una lista de argumentos. La diferencia es que `printf` envía la salida con formato directamente al flujo de salida estándar, mientras que `format` envía la salida con formato al destino especificado por su constructor (en este programa, una cadena en la memoria). En la línea 15 se invoca el método `toString` de `Formatter` para obtener los datos con formato, como una cadena que luego se muestra en un cuadro de diálogo de mensaje.

Observe que la clase `String` también proporciona un método de conveniencia `static` llamado `format`, el cual nos permite crear una cadena en la memoria, sin necesidad de crear primero un objeto `Formatter`. Podríamos haber sustituido las líneas 11 a 12 y la línea 15 de la figura 29.24 por:

```
String s = String.format( "%d = %#o = %#^x", 10, 10, 10 );
JOptionPane.showMessageDialog( null, s );
```

```
1  // Fig. 29.24: PruebaFormatter.java
2  // Cadena de formato con la clase Formatter.
3  import java.util.Formatter;
4  import javax.swing.JOptionPane;
5
6  public class PruebaFormatter
7  {
8     public static void main( String args[] )
9     {
10       // crea un objeto Formatter y aplica formato a la salida
11       Formatter formatter = new Formatter();
12       formatter.format( "%d = %#o = %#X", 10, 10, 10 );
13
14       // muestra la salida en el componente JOptionPane
15       JOptionPane.showMessageDialog( null, formatter.toString() );
16    } // fin de main
17 } // fin de la clase PruebaFormatter
```



Figura 29.24 | Aplicar formato a la salida con la clase `Formatter`.

29.14 Conclusión

En este capítulo vimos un resumen acerca de cómo aplicar formato a la salida mediante los diversos caracteres y banderas de formato. Mostramos números decimales mediante el uso de los caracteres de formato `d`, `o`, `x` y `X`. Mostramos números de punto flotante usando los caracteres de formato `e`, `E`, `f`, `g` y `G`. Mostramos la fecha y la hora en diversos formatos, usando los caracteres de formato `t` y `T` junto con sus caracteres de sufijo de conversión. Aprendió a mostrar la salida con anchuras de campo y precisiones. Presentamos las banderas `+`, `-`, espacio, `#`, `0`, coma y `(` que se utilizan en conjunto con los caracteres de formato para producir la salida. También demostramos

cómo aplicar formato a la salida con la clase `Formatter`. En el siguiente capítulo hablaremos sobre los métodos de la clase `String` para manipular cadenas. También presentaremos las expresiones regulares y demostraremos cómo validar la entrada del usuario mediante éstas.

Resumen

Sección 29.2 Flujos

- Por lo general, las operaciones de entrada y salida se llevan a cabo con flujos, los cuales son secuencias de bytes. En las operaciones de entrada, los bytes fluyen de un dispositivo a la memoria principal. En las operaciones de salida, los bytes fluyen de la memoria principal a un dispositivo.
- Por lo común, el flujo de entrada estándar se conecta al teclado, y el flujo de salida estándar se conecta a la pantalla de la computadora.

Sección 29.3 Aplicación de formato a la salida con `printf`

- La cadena de formato de `printf` describe los formatos en los que aparecen los valores de salida. El especificador de formato consiste en un índice como argumento, banderas, anchuras de campo, precisiones y caracteres de conversión.

Sección 29.4 Impresión de enteros

- Los enteros se imprimen con los caracteres de conversión `d` para enteros decimales, `o` para enteros en formato octal y `x` (o `X`) para los enteros en formato hexadecimal. Cuando se utiliza el carácter de conversión `X`, la salida se muestra en letras mayúsculas.

Sección 29.5 Impresión de números de punto flotante

- Los valores de punto flotante se imprimen con los caracteres de conversión `e` (o `E`) para la notación exponencial, `f` para la notación de punto flotante regular, y `g` (o `G`) para la notación `e` (o `E`) o `f`. Cuando se indica el especificador de conversión `g`, se utiliza el carácter de conversión `e` si el valor es menor que 10^{-3}, o mayor o igual a 10^7; en caso contrario, se utiliza el carácter de conversión `f`. Cuando se utilizan los caracteres de conversión `E` y `G`, la salida se muestra en letras mayúsculas.

Sección 29.6 Impresión de cadenas y caracteres

- El carácter de conversión `c` imprime un carácter.
- El carácter de conversión `s` (o `S`) imprime una cadena de caracteres. Cuando se utiliza el carácter de conversión `S`, la salida se muestra en letras mayúsculas.

Sección 29.7 Impresión de fechas y horas

- El carácter de conversión `t` (o `T`) seguido de un carácter de sufijo de conversión imprime la fecha y la hora en diversos formatos. Cuando se utiliza el carácter de conversión `T`, la salida se muestra en letras mayúsculas.
- El carácter de conversión `t` (o `T`) requiere que el argumento sea de tipo `long`, `Long`, `Calendar` o `Date`.

Sección 29.8 Otros caracteres de conversión

- El carácter de conversión `b` (o `B`) imprime la representación de cadena de un valor `boolean` o `Boolean`. Estos caracteres de conversión también imprimen `"true"` para las referencias que no son `null`, y `"false"` para las referencias `null`. Cuando se utiliza el carácter de conversión `B`, la salida se muestra en letras mayúsculas.
- El carácter de conversión `h` (o `H`) devuelve `null` para una referencia `null`, y una representación `String` del valor de código hash (en base 16) del objeto. Los códigos de hash se utilizan para almacenar y obtener objetos que se encuentran en objetos `Hashtable` y `HashMap`. Cuando se utiliza el carácter de conversión `H`, la salida se muestra en letras mayúsculas.
- El carácter de conversión `n` imprime el separador de línea específico de la plataforma.
- El carácter de conversión `%` se utiliza para mostrar un `%` literal.

Sección 29.9 Impresión con anchuras de campo y precisiones

- Si la anchura de campo es mayor que el objeto a imprimir, éste se justifica a la derecha en el campo.

- Las anchuras de campo se pueden usar con todos los caracteres de conversión, excepto la conversión con el separador de línea.
- La precisión que se utiliza con los caracteres de conversión de punto flotante `e` y `f` indica el número de dígitos que aparecen después del punto decimal. La precisión que se utiliza con el carácter de conversión de punto flotante `g` indica el número de dígitos significativos que deben aparecer.
- La precisión que se utiliza con el carácter de conversión `s` indica el número de caracteres a imprimir.
- La anchura de campo y la precisión se pueden combinar, para lo cual se coloca la anchura de campo, seguida de un punto decimal, seguido de la precisión entre el signo de porcentaje y el carácter de conversión.

Sección 29.10 Uso de banderas en la cadena de formato de `printf`

- La bandera – justifica a la izquierda su argumento en un campo.
- La bandera `+` imprime un signo más para los valores positivos, y un signo menos para los valores negativos.
- La bandera de espacio imprime un espacio antes de un valor positivo. La bandera de espacio y la bandera `+` no se pueden utilizar juntas en un carácter de conversión integral.
- La bandera `#` antepone un `0` a los valores octales, y `0x` a los valores hexadecimales.
- La bandera `0` imprime ceros a la izquierda para un valor que no ocupa todo su campo.
- La bandera de coma (`,`) utiliza el separador de miles específico para la configuración regional (por ejemplo, `','` para los EUA), para mostrar números enteros y de punto flotante.
- La bandera `(` encierra un número negativo entre paréntesis.

Sección 29.11 Impresión con índices como argumentos

- Un índice como argumento es un entero decimal opcional, seguido de un signo `$` que indica la posición del argumento en la lista de argumentos.
- Los índices como argumentos permiten a los programadores reordenar la salida, de manera que los argumentos en la lista de argumentos no estén necesariamente en el orden de sus correspondientes especificadores de formato. Los índices como argumentos también ayudan a evitar los argumentos duplicados.

Sección 29.13 Aplicación de formato a la salida con la clase `Formatter`

- La clase `Formatter` (en el paquete `java.util`) proporciona las mismas herramientas de formato que `printf`. `Formatter` es una clase utilitaria que permite a los programadores imprimir salida con formato hacia varios destinos, incluyendo componentes de GUI, archivos y otros flujos de salida.
- El método `format` de la clase `Formatter` imprime los datos con formato al destino especificado por el constructor de `Formatter`.
- El método `static format` de la clase `String` aplica formato a los datos y devuelve los datos con formato, como un objeto `String`.

Terminología

`#`, bandera
`%`, carácter de conversión
`(`, bandera
`+` (más), bandera
`-` (menos), bandera
`,` (coma), bandera
`0` (cero), bandera
`A`, carácter de sufijo de conversión
`a`, carácter de sufijo de conversión
alineación
anchura de campo
`b`, carácter de conversión
`B`, carácter de conversión
`B`, carácter de sufijo de conversión
`b`, carácter de sufijo de conversión
bandera
bandera de espacio
`c`, carácter de conversión
`C`, carácter de sufijo de conversión
cadena de formato
carácter de conversión
conversión de enteros
conversión de números de punto flotante
`d`, carácter de conversión
`D`, carácter de sufijo de conversión
`e`, carácter de conversión
`E`, carácter de conversión
`e`, carácter de sufijo de conversión
especificador de formato
`f`, carácter de conversión
`F`, carácter de sufijo de conversión
flujo
flujo de entrada estándar
flujo de error estándar

flujo de salida estándar
`format`, método de `Formatter`
`format`, método de `String`
formato de punto flotante exponencial
formato hexadecimal
formato octal
`Formatter`, clase
`g`, carácter de conversión
`G`, carácter de conversión
`h`, carácter de conversión
`H`, carácter de conversión
`H`, carácter de sufijo de conversión
`I`, carácter de sufijo de conversión
índice como argumento
`j`, carácter de sufijo de conversión
justificación a la derecha
justificación a la izquierda
`K`, carácter de sufijo de conversión
`m`, carácter de sufijo de conversión
`M`, carácter de sufijo de conversión
`n`, carácter de conversión
notación científica
`o`, carácter de conversión
`P`, carácter de sufijo de conversión
`p`, carácter de sufijo de conversión
precisión
`printf`, método
punto flotante
`r`, carácter de sufijo de conversión
redirigir un flujo
redondeo
`s`, carácter de conversión
`S`, carácter de conversión
`S`, carácter de sufijo de conversión
`t`, carácter de conversión
`T`, carácter de conversión
`T`, carácter de sufijo de conversión
`toString`, método de `Formatter`
`x`, carácter de conversión
`y`, carácter de sufijo de conversión
`Y`, carácter de sufijo de conversión
`Z`, carácter de sufijo de conversión

Ejercicios de autoevaluación

29.1 Complete los enunciados:

a) Todas las operaciones de entrada y salida se manejan en forma de ______________.
b) El flujo ______________ se conecta generalmente al teclado.
c) El flujo ______________ se conecta generalmente a la pantalla de la computadora.
d) El método ______________ de `System.out` se puede utilizar para aplicar formato al texto que se muestra en la salida estándar.
e) El carácter de conversión ______________ puede utilizarse para imprimir en pantalla un entero decimal.
f) Los caracteres de conversión ______________ y ______________ se utilizan para mostrar enteros en formato octal y hexadecimal, respectivamente.
g) El carácter de conversión ______________ se utiliza para mostrar un valor de punto flotante en notación exponencial
h) Los caracteres de conversión `e` y `f` se muestran con ______________ dígitos de precisión a la derecha del punto decimal, si no se especifica una precisión.
i) Los caracteres de conversión ______________ y ______________ se utilizan para imprimir cadenas y caracteres, respectivamente.
j) El carácter de conversión ______________ y el carácter de sufijo de conversión ______________ se utilizan para imprimir la hora para el reloj de 24 horas, como `hora:minuto:segundo`.
k) La bandera ______________ hace que la salida se justifique a la izquierda en un campo.
l) La bandera ______________ hace que los valores se muestren con un signo más o con un signo menos.
m) El índice como argumento ______________ corresponde al segundo argumento en la lista de argumentos.
n) La clase ______________ tiene la misma capacidad que `printf`, pero permite a los programadores imprimir salida con formato en varios destinos, además del flujo de salida estándar.

29.2 Encuentre el error en cada uno de los siguientes enunciados, y explique cómo se puede corregir.

a) La siguiente instrucción debe imprimir el carácter `'c'`.
```
System.out.printf( "%c\n", "c" );
```
b) La siguiente instrucción debe imprimir `9.375%`.
```
System.out.printf( "%.3f%", 9.375 );
```

c) La siguiente instrucción debe imprimir el tercer argumento en la lista de argumentos:
`System.out.printf( "%2$s\n", "Lun", "Mar", "Mie", "Jue", "Vie" );`

d) `System.out.printf( ""Una cadena entre comillas"");`

e) `System.out.printf( %d %d, 12, 20 );`

f) `System.out.printf( "%s\n", 'Richard' );`

29.3 Escriba una instrucción para cada uno de los siguientes casos:

a) Imprimir 1234 justificado a la derecha, en un campo de 10 dígitos.

b) Imprimir 123.456789 en notación exponencial con un signo (+ o -) y 3 dígitos de precisión.

c) Imprimir 100 en formato octal, precedido por 0.

d) Dado un objeto Calendario de la clase Calendar, imprima una fecha con formato de mes/día/año (cada uno con dos dígitos).

e) Dado un objeto Calendar llamado calendario, imprimir una hora para el reloj de 24 horas como hora:minuto:segundo (cada uno con dos dígitos), usando un índice como argumento y caracteres de sufijo de conversión para aplicar formato a la hora.

f) Imprimir 3.333333 con un signo (+ o -) en un campo de 20 caracteres, con una precisión de 3.

Respuestas a los ejercicios de autoevaluación

29.1 a) Flujos. b) de entrada estándar. c) de salida estándar. d) printf. e) d. f) o, x o X. g) e o E. h) 6. i) s o S, c o C. j) t, T. k) – (menos). l) + (más). m) 2$. n) Formatter.

29.2 a) Error: el carácter de conversión c espera un argumento del tipo primitivo char.
Corrección: para imprimir el carácter 'c', cambie "c" a 'c'.

b) Error: está tratando de imprimir el carácter literal % sin usar el especificador de formato %%.
Corrección: use %% para imprimir un carácter % literal.

c) Error: el índice como argumento no empieza con 0; por ejemplo, el primer argumento es 1$.
Corrección: para imprimir el tercer argumento, use 3$.

d) Error: está tratando de imprimir el carácter literal " sin usar la secuencia de escape \".
Corrección: sustituya cada comilla en el conjunto interno de comillas con \".

e) Error: la cadena de formato no va encerrada entre comillas dobles.
Corrección: encierre %d %d entre comillas dobles.

f) Error: la cadena a imprimir está encerrada entre comillas.
Corrección: use dobles comillas en vez de comillas sencillas para representar una cadena.

29.3 a) `System.out.printf( "%10d\n", 1234 );`

b) `System.out.printf( "%+.3e\n", 123.456789 );`

c) `System.out.printf( "%#o\n", 100 );`

d) `System.out.printf( "%tD\n", calendario );`

e) `System.out.printf( "%1$tH:%1$tM:%1$tS\n", calendario );`

f) `System.out.printf( "%+20.3f\n", 3.333333 );`

Ejercicios

29.4 Escriba una o más instrucciones para cada uno de los siguientes casos:

a) Imprimir el entero 40000 justificado a la derecha en un campo de 15 dígitos.

b) Imprimir 200 con y sin un signo.

c) Imprimir 100 en formato hexadecimal, precedido por 0x.

d) Imprimir 1.234 con tres dígitos de precisión en un campo de nueve dígitos con ceros a la izquierda.

29.5 Muestre lo que se imprime en cada una de las siguientes instrucciones. Si una instrucción es incorrecta, indique por qué.

a) `System.out.printf( "%-10d\n", 10000 );`

b) `System.out.printf( "%c\n", "Esta es una cadena" );`

c) `System.out.printf( "%8.3f\n", 1024.987654 );`

d) `System.out.printf( "%#o\n%#X\n", 17, 17 );`

e) `System.out.printf( "% d\n%+d\n", 1000000, 1000000 );`

f) `System.out.printf( "%10.2e\n", 444.93738 );`

g) `System.out.printf( "%d\n", 10.987 );`

29.6 Encuentre el(los) error(es) en cada uno de los siguientes segmentos de programa. Muestre la instrucción corregida.

a) `System.out.printf( "%s\n", 'Feliz cumpleanios' );`

b) `System.out.printf( "%c\n", 'Hola' );`

c) `System.out.printf( "%c\n", "Esta es una cadena" );`

d) La siguiente instrucción debe imprimir `"Buen viaje"` con las dobles comillas:
`System.out.printf( ""%s"", "Buen viaje" );`

e) La siguiente instrucción debe imprimir `"Hoy es viernes"`:
`System.out.printf( "Hoy es %s\n", "Lunes", "Viernes" );`

f) `System.out.printf( 'Escriba su nombre: ' );`

g) `System.out.printf( %f, 123.456 );`

h) La siguiente instrucción debe imprimir la hora actual en el formato `"hh:mm:ss"`:

```
Calendar fechaHora = Calendar.getInstance();
System.out.printf( "%1$tk:1$%tl:%1$tS\n", fechaHora );
```

29.7 *(Impresión de fechas y horas)* Escriba un programa que imprima fechas y horas en los siguientes formatos:

```
GMT-05:00 04/30/04 09:55:09 AM
GMT-05:00 Abril 30 2004 09:55:09
2004-04-30 dia-del-mes:30
2004-04-30 dia-del-anio: 121
Vie Abr 30 09:55:09 GMT-05:00  2004
```

[*Nota:* dependiendo de su ubicación, tal vez tenga una zona horaria distinta de GMT-05:00].

29.8 Escriba un programa para probar los resultados de imprimir el valor entero `12345` y el valor de punto flotante `1.2345` en campos de varios tamaños.

29.9 *(Redondeo de números)* Escriba un programa que imprima el valor `100.453267` redondeado al dígito, a la decena, centena, múltiplo de mil y de diez mil más cercanos.

29.10 Escriba un programa que reciba como entrada una palabra del teclado y determine su longitud. Imprima la palabra usando el doble de la longitud como anchura del campo.

29.11 *(Conversión de temperatura en grados Fahrenheit a Centígrados)* Escriba un programa que convierta temperaturas enteras en grados Fahrenheit de `0` a `212` grados, a temperaturas en grados Centígrados de punto flotante con tres dígitos de precisión. Use la siguiente fórmula:

```
centigrados = 5.0 / 9.0 * ( fahrenheit - 32 );
```

para realizar el cálculo. La salida debe imprimirse en dos columnas justificadas a la derecha de 10 caracteres cada una, y las temperaturas en grados Centígrados deben ir precedidas por un signo, tanto para los valores positivos como para los negativos.

29.12 Escriba un programa para probar todas las secuencias de escape en la figura 29.23. Para las secuencias de escape que desplazan el cursor, imprima un carácter antes y después de la secuencia de escape, de manera que se pueda ver con claridad a dónde se ha desplazado el cursor.

29.13 Escriba un programa que utilice el carácter de conversión `g` para imprimir el valor `9876.12345`. Imprima el valor con precisiones que varíen de `1` a `9`.

30



El principal defecto del rey Enrique era masticar pequeños pedazos de hilo.

—Hilaire Belloc

La escritura vigorosa es concisa. Una oración no debe contener palabras innecesarias, un párrafo no debe contener oraciones innecesarias.

—William Strunk, Jr.

He extendido esta carta más de lo usual, ya que carezco de tiempo para hacerla breve.

—Blaise Pascal

Cadenas, caracteres y expresiones regulares

OBJETIVOS

En este capítulo aprenderá a:

- Crear y a manipular objetos de cadena de caracteres inmutables de la clase `String`.
- Crear y a manipular objetos de cadena de caracteres mutables de la clase `StringBuilder`.
- Crear y a manipular objetos de la clase `Character`.
- Usar un objeto `StringTokenizer` para dividir un objeto `String` en tokens.
- Usar expresiones regulares para validar los datos `String` introducidos en una aplicación.

Plan general

30.1 Introducción

En este capítulo presentamos las herramientas para el procesamiento de cadenas y caracteres en Java. Las técnicas que se describen aquí son apropiadas para validar la entrada en los programas, para mostrar información a los usuarios y otras manipulaciones basadas en texto. Estas técnicas son también apropiadas para desarrollar editores de texto, procesadores de palabras, software para el diseño de páginas, sistemas de composición computarizados y demás tipos de software para el procesamiento de texto. Ya hemos presentado varias herramientas para el procesamiento de texto en capítulos anteriores. En este capítulo describiremos detalladamente las herramientas de las clases `String`, `StringBuilder` y `Character` del paquete `java.lang`, y la clase `StringTokenizer` del paquete `java.util`. Estas clases proporcionan la base para la manipulación de cadenas y caracteres en Java.

En este capítulo también hablaremos sobre las expresiones regulares que proporcionan a las aplicaciones la capacidad de validar los datos de entrada. Esta funcionalidad se encuentra en la clase `String`, junto con las clases `Matcher` y `Pattern` ubicadas en el paquete `java.util.regex`.

30.2 Fundamentos de los caracteres y las cadenas

Los caracteres son los bloques de construcción básicos de los programas fuente de Java. Todo programa está compuesto de una secuencia de caracteres que, cuando se agrupan en forma significativa, son interpretados por la computadora como una serie de instrucciones utilizadas para realizar una tarea. Un programa puede contener **literales de carácter**. Una literal de carácter es un valor entero representado como un carácter entre comillas simples. Por ejemplo, `'z'` representa el valor entero de `z`, y `'\n'` representa el valor entero de una nueva línea. El valor de una literal de carácter es el valor entero del carácter en el **conjunto de caracteres Unicode**. En el apéndice B se presentan los equivalentes enteros de los caracteres en el conjunto de caracteres ASCII, el cual es un subconjunto de Unicode (que veremos en el apéndice I). Para obtener información detallada sobre Unicode, visite `www.unicode.org`.

Recuerde que en la sección 2.2 vimos que una cadena es una secuencia de caracteres que se trata como una sola unidad. Una cadena puede incluir letras, dígitos y varios **caracteres especiales**, tales como `+`, `-`, `*`, `/` y `$`. Una cadena es un objeto de la clase `String`. Las **literales de cadena** (que se almacenan en memoria como objetos `String`) se escriben como una secuencia de caracteres entre comillas dobles, como en:

`"Juan E. Pérez"`	(un nombre)
`"Calle Principal 9999"`	(una dirección)
`"Monterrey, Nuevo León"`	(una ciudad y un estado)
`"(201) 555-1212"`	(un número telefónico)

Una cadena puede asignarse a una referencia `String`. La declaración

```
String color = "azul";
```

inicializa la variable `String` de nombre `color` para que haga referencia a un objeto `String` que contiene la cadena `"azul"`.



Tip de rendimiento 30.1

Java trata a todas las literales de cadena con el mismo contenido como un solo objeto `String` que tiene muchas referencias. Esto ayuda a conservar memoria.

30.3 La clase String

La clase `String` se utiliza para representar cadenas en Java. En las siguientes subsecciones cubriremos muchas de las herramientas de la clase `String`.

30.3.1 Constructores de String

La clase `String` proporciona constructores para inicializar objetos `String` de varias formas. Cuatro de los constructores se muestran en el método `main` de la figura 30.1.

En la línea 12 se crea una instancia de un nuevo objeto `String`, utilizando el constructor sin argumentos de la clase `String`, y se le asigna su referencia a `s1`. El nuevo objeto `String` no contiene caracteres (la **cadena vacía**) y tiene una longitud de 0.

En la línea 13 se crea una instancia de un nuevo objeto `String`, utilizando el constructor de la clase `String` que toma un objeto `String` como argumento, y se le asigna su referencia a `s2`. El nuevo objeto `String` contiene la misma secuencia de caracteres que el objeto `String` de nombre `s`, el cual se pasa como argumento para el constructor.



Observación de ingeniería de software 30.1

No es necesario copiar un objeto `String` existente. Los objetos `String` son inmutables: su contenido de caracteres no puede modificarse una vez que son creados, ya que la clase `String` no proporciona métodos que permitan modificar el contenido de un objeto `String`.

En la línea 14 se crea la instancia de un nuevo objeto `String` y se le asigna su referencia a `s3`, utilizando el constructor de la clase `String` que toma un arreglo de caracteres como argumento. El nuevo objeto `String` contiene una copia de los caracteres en el arreglo.

En la línea 15 se crea la instancia de un nuevo objeto `String` y se le asigna su referencia a `s4`, utilizando el constructor de la clase `String` que toma un arreglo `char` y dos enteros como argumentos. El segundo argumento especifica la posición inicial (el desplazamiento) a partir del cual se accede a los caracteres en el arreglo. Recuerde que el primer carácter se encuentra en la posición `0`. El tercer argumento especifica el número de caracteres (la cuenta) que se van a utilizar del arreglo. El nuevo objeto `String` contiene una cadena formada a partir de los caracteres utilizados. Si el desplazamiento o la cuenta especificados como argumentos ocasionan que se acceda a un elemento fuera de los límites del arreglo de caracteres, se lanza una excepción `StringIndexOutOfBoundsException`.

```
// Fig. 30.1: ConstructoresString.java
// Constructores de la clase String.

public class ConstructoresString
{
   public static void main( String args[] )
   {
      char arregloChar[] = { 'c', 'u', 'm', 'p', 'l', 'e', ' ', 'a', 'n', 'i', 'o', 's' };
      String s = new String( "hola" );

      // usa los constructores de String
      String s1 = new String();
      String s2 = new String( s );
      String s3 = new String( arregloChar );
      String s4 = new String( arregloChar, 7, 5 );

      System.out.printf(
         "s1 = %s\ns2 = %s\ns3 = %s\ns4 = %s\n",
         s1, s2, s3, s4 ); // muestra las cadenas
   } // fin de main
} // fin de la clase ConstructoresString
```

```
s1 =
s2 = hola
s3 = cumple anios
s4 = anios
```

Figura 30.1 | Constructores de la clase `String`.



Error común de programación 30.1

Intentar acceder a un carácter que se encuentra fuera de los límites de una cadena (es decir, un índice menor que 0 o un índice mayor o igual a la longitud de la cadena), se produce una excepción `StringIndexOutOfBoundsException`.

30.3.2 Métodos `length`, `charAt` y `getChars` de `String`

Los métodos `length`, `charAt` y `getChars` de `String` determinan la longitud de una cadena, obtienen el carácter que se encuentra en una ubicación específica de una cadena y recuperan el conjunto completo de caracteres en una cadena, respectivamente. La aplicación de la figura 30.2 muestra cada uno de estos métodos.

```
// Fig. 30.2: VariosString.java
// Esta aplicación muestra los métodos length, charAt y getChars
// de la clase String.

public class VariosString
{
   public static void main( String args[] )
   {
      String s1 = "hola a todos";
      char arregloChar[] = new char[ 5 ];

      System.out.printf( "s1: %s", s1 );

      // prueba el método length
      System.out.printf( "\nLongitud de s1: %d", s1.length() );
```

Figura 30.2 | Métodos de manipulación de caracteres de la clase `String`. (Parte 1 de 2).

```
16
17         // itera a través de los caracteres en s1 con charAt y muestra la cadena invertida
18         System.out.print( "\nLa cadena invertida es: " );
19
20         for ( int cuenta = s1.length() - 1; cuenta >= 0; cuenta-- )
21            System.out.printf( "%s ", s1.charAt( cuenta) );
22
23         // copia los caracteres de la cadena a arregloChar
24         s1.getChars( 0, 4, arregloChar, 0 );
25         System.out.print( "\nEl arreglo de caracteres es: " );
26
27         for ( char caracter : arregloChar )
28            System.out.print( caracter );
29
30         System.out.println();
31      } // fin de main
32   } // fin de la clase VariosString
```

```
s1: hola a todos
Longitud de s1: 12
La cadena invertida es: s o d o t   a   a l o h
El arreglo de caracteres es: hola
```

Figura 30.2 | Métodos de manipulación de caracteres de la clase `String`. (Parte 2 de 2).

En la línea 15 se utiliza el método `length` de `String` para determinar el número de caracteres en la cadena `s1`. Al igual que los arreglos, las cadenas conocen su propia longitud. Sin embargo, a diferencia de los arreglos, no podemos acceder a la longitud de una cadena mediante un campo `length`; en vez de ello, debemos llamar al método `length` del objeto `String`.

En las líneas 20 y 21 se imprimen los caracteres de la cadena `s1` en orden inverso (y separados por espacios). El método `charAt` de `String` (línea 21) devuelve el carácter ubicado en una posición específica en la cadena. El método `charAt` recibe un argumento entero que se utiliza como el índice, y devuelve el carácter en esa posición. Al igual que los arreglos, se considera que el primer elemento de una cadena está en la posición 0.

En la línea 24 se utiliza el método `getChars` de `String` para copiar los caracteres de una cadena en un arreglo de caracteres. El primer argumento es el índice inicial en la cadena, a partir del cual se van a copiar los caracteres. El segundo argumento es el índice que está una posición más adelante del último carácter que se va a copiar de la cadena. El tercer argumento es el arreglo de caracteres en el que se van a copiar los caracteres. El último argumento es el índice inicial en donde se van a colocar los caracteres copiados en el arreglo de caracteres de destino. A continuación, en la línea 28 se imprime el contenido del arreglo `char`, un carácter a la vez.

30.3.3 Comparación entre cadenas

En el capítulo 7 hablamos sobre el ordenamiento y la búsqueda en los arreglos. Con frecuencia, la información que se va a ordenar o buscar consiste en cadenas que deben compararse para determinar el orden o para determinar si una cadena aparece en un arreglo (u otra colección). La clase `String` proporciona varios métodos para comparar cadenas, los cuales mostraremos en los siguientes dos ejemplos.

Para comprender lo que significa que una cadena sea mayor o menor que otra, considere el proceso de alfabetizar una serie de apellidos. Sin duda usted colocaría a "Jones" antes que "Smith", ya que en el alfabeto la primera letra de "Jones" viene antes que la primera letra de "Smith" en el alfabeto. Pero el alfabeto es algo más que una lista de 26 letras; es un conjunto ordenado de caracteres. Cada letra ocupa una posición específica dentro del conjunto. Z es más que una letra del alfabeto; es en específico la letra número veintiséis del alfabeto.

¿Cómo sabe la computadora que una letra va antes que otra? Todos los caracteres se representan en la computadora como códigos numéricos (vea el apéndice B). Cuando la computadora compara cadenas, en realidad compara los códigos numéricos de los caracteres en las cadenas.

En la figura 30.3 se muestran los métodos `equals`, `equalsIgnoreCase`, `compareTo` y **`regionMatches`** de `String`, y se muestra el uso del operador de igualdad == para comparar objetos `String`.

```
// Fig. 30.3: CompararCadenas.java
// Los métodos equals, equalsIgnoreCase, compareTo y regionMatches de String.

public class CompararCadenas
{
   public static void main( String args[] )
   {
      String s1 = new String( "hola" ); // s1 es una copia de "hola"
      String s2 = "adios";
      String s3 = "Feliz cumpleanios";
      String s4 = "feliz cumpleanios";

      System.out.printf(
         "s1 = %s\ns2 = %s\ns3 = %s\ns4 = %s\n\n", s1, s2, s3, s4 );

      // prueba la igualdad
      if ( s1.equals( "hola" ) )  // true
         System.out.println( "s1 es igual a \"hola\"" );
      else
         System.out.println( "s1 no es igual a \"hola\"" );

      // prueba la igualdad con ==
      if ( s1 == "hola" )  // false; no son el mismo objeto
         System.out.println( "s1 es el mismo objeto que \"hola\"" );
      else
         System.out.println( "s1 no es el mismo objeto que \"hola\"" );

      // prueba la igualdad (ignora el uso de mayúsculas/minúsculas)
      if ( s3.equalsIgnoreCase( s4 ) )  // true
         System.out.printf( "%s es igual a %s si se ignora el uso de mayusculas/
                           minusculas\n", s3, s4 );
      else
         System.out.println( "s3 no es igual a s4" );

      // prueba con compareTo
      System.out.printf(
         "\ns1.compareTo( s2 ) es %d", s1.compareTo( s2 ) );
      System.out.printf(
         "\ns2.compareTo( s1 ) es %d", s2.compareTo( s1 ) );
      System.out.printf(
         "\ns1.compareTo( s1 ) es %d", s1.compareTo( s1 ) );
      System.out.printf(
         "\ns3.compareTo( s4 ) es %d", s3.compareTo( s4 ) );
      System.out.printf(
         "\ns4.compareTo( s3 ) es %d\n\n", s4.compareTo( s3 ) );

      // prueba con regionMatches (sensible a mayúsculas/minúsculas)
      if ( s3.regionMatches( 0, s4, 0, 5 ) )
         System.out.println( "Los primeros 5 caracteres de s3 y s4 coinciden" );
      else
         System.out.println(
            "Los primeros 5 caracteres de s3 y s4 no coinciden" );

      // prueba con regionMatches (ignora el uso de mayúsculas/minúsculas)
      if ( s3.regionMatches( true, 0, s4, 0, 5 ) )
         System.out.println( "Los primeros 5 caracteres de s3 y s4 coinciden" );
      else
         System.out.println(
            "Los primeros 5 caracteres de s3 y s4 no coinciden" );
```

Figura 30.3 | Comparaciones entre objetos `String`. (Parte 1 de 2).

```
59     } // fin de main
60  } // fin de la clase CompararCadenas
```

```
s1 = hola
s2 = adios
s3 = Feliz cumpleanios
s4 = feliz cumpleanios

s1 es igual a "hola"
s1 no es el mismo objeto que "hola"
Feliz cumpleanios es igual a feliz cumpleanios si se ignora el uso de mayusculas
/minusculas

s1.compareTo( s2 ) es 7
s2.compareTo( s1 ) es -7
s1.compareTo( s1 ) es 0
s3.compareTo( s4 ) es -32
s4.compareTo( s3 ) es 32

Los primeros 5 caracteres de s3 y s4 no coinciden
Los primeros 5 caracteres de s3 y s4 coinciden
```

Figura 30.3 | Comparaciones entre objetos `String`. (Parte 2 de 2).

La condición en la línea 17 utiliza el método `equals` para comparar la igualdad entre la cadena `s1` y la literal de cadena `"hola"`. El método `equals` (un método de la clase `Object`, sobrescrito en `String`) prueba la igualdad entre dos objetos (es decir, que las cadenas contenidas en los dos objetos sean idénticas). El método devuelve `true` si el contenido de los objetos es igual y `false` en caso contrario. La condición anterior es `true`, ya que la cadena `s1` fue inicializada con la literal de cadena `"hola"`. El método `equals` utiliza una **comparación lexicográfica**; se comparan los valores enteros Unicode (vea el apéndice I, Unicode®, en el sitio Web www.deitel.com/jhtp7) que representan a cada uno de los caracteres en cada cadena. Por lo tanto, si la cadena `"hola"` se compara con la cadena `"HOLA"`, el resultado es `false` ya que la representación entera de una letra minúscula es distinta de la letra mayúscula correspondiente.

La condición en la línea 23 utiliza el operador de igualdad == para comparar la igualdad entre la cadena `s1` y la literal de cadena `"hola"`. El operador == tiene distinta funcionalidad cuando se utiliza para comparar referencias, que cuando se utiliza para comparar valores de tipos primitivos. Cuando se comparan valores de tipos primitivos con ==, el resultado es `true` si ambos valores son idénticos. Cuando se comparan referencias con `==`, el resultado es `true` si ambas referencias se refieren al mismo objeto en memoria. Para comparar la igualdad del contenido en sí (o información sobre el estado) de los objetos, hay que invocar un método. En el caso de los objetos `String`, ese método es `equals`. La condición anterior se evalúa como `false` en la línea 23, ya que la referencia `s1` se inicializó con la instrucción

```
s1 = new String( "hola" );
```

con lo cual se crea un nuevo objeto `String` con una copia de la literal de cadena `"hola"`, y se asigna el nuevo objeto a la variable `s1`. Si `s1` se hubiera inicializado con la instrucción

```
s1 = "hola";
```

con lo cual se asigna directamente la literal de cadena `"hola"` a la variable `s1`, la condición hubiera sido `true`. Recuerde que Java trata a todos los objetos de literal de cadena con el mismo contenido como un objeto `String`, al cual puede haber muchas referencias. Por lo tanto, en las líneas 8, 17 y 23 se hace referencia al mismo objeto `String` `"hola"` en memoria.



Error común de programación 30.2

Al comparar referencias con `==` *se pueden producir errores lógicos, ya que* `==` *compara las referencias para determinar si se refieren al mismo objeto, no si dos objetos tienen el mismo contenido. Si se comparan dos objetos idénticos (pero separados) con* `==`*, el resultado será* `false`*. Si va a comparar objetos para determinar si tienen el mismo contenido, utilice el método* `equals`*.*

Si va a ordenar objetos `String`, puede comparar si son iguales con el método `equalsIgnoreCase`, el cual ignora si las letras en cada cadena son mayúsculas o minúsculas al realizar la comparación. Por lo tanto, la cadena `"hola"` y la cadena `"HOLA"` se consideran iguales. En la línea 29 se utiliza el método `equalsIgnoreCase` de `String` para comparar si la cadena `s3` (`Feliz Cumpleanios`) es igual a la cadena `s4` (`feliz cumpleanios`). El resultado de esta comparación es `true`, ya que la comparación ignora el uso de mayúsculas y minúsculas.

En las líneas 35 a 44 se utiliza el método `compareTo` de `String` para comparar cadenas. El método `compareTo` se declara en la interfaz `Comparable` y se implementa en la clase `String`. En la línea 36 se compara la cadena `s1` con la cadena `s2`. El método `compareTo` devuelve 0 si las cadenas son iguales, un número negativo si la cadena que invoca a `compareTo` es menor que la cadena que se pasa como argumento, y un número positivo si la cadena que invoca a `compareTo` es mayor que la cadena que se pasa como argumento. El método `compareTo` utiliza una comparación lexicográfica; compara los valores numéricos de los caracteres correspondientes en cada cadena (para obtener más información acerca del valor exacto devuelto por el método `compareTo`, vea la página `java.sun.com/javase/6/docs/api/java/lang/String.html`).

La condición en la línea 47 utiliza el método `regionMatches` de `String` para comparar si ciertas porciones de dos cadenas son iguales. El primer argumento es el índice inicial en la cadena que invoca al método. El segundo argumento es una cadena de comparación. El tercer argumento es el índice inicial en la cadena de comparación. El último argumento es el número de caracteres a comparar entre las dos cadenas. El método devuelve `true` solamente si el número especificado de caracteres son lexicográficamente iguales.

Por último, la condición en la línea 54 utiliza una versión con cinco argumentos del método `regionMatches` de `String` para comparar si ciertas porciones de dos cadenas son iguales. Cuando el primer argumento es `true`, el método ignora el uso de mayúsculas y minúsculas en los caracteres que se van a comparar. Los argumentos restantes son idénticos a los que se describieron para el método `regionMatches` con cuatro argumentos.

En el siguiente ejemplo (figura 30.4) vamos a mostrar los métodos **`startsWith`** y **`endsWith`** de la clase `String`. El método `main` crea el arreglo `cadenas` que contiene las cadenas `"empezo"`, `"empezando"`, `"termino"` y `"terminando"`. El resto del método `main` consiste en tres instrucciones `for` que prueban los elementos del arreglo para determinar si empiezan o terminan con un conjunto específico de caracteres.

```java
// Fig. 30.4: CadenaInicioFin.java
// Los métodos startsWith y endsWith de String.

public class CadenaInicioFin
{
   public static void main( String args[] )
   {
      String cadenas[] = { "empezo", "empezando", "termino", "terminando" };

      // prueba el método startsWith
      for ( String cadena: cadenas )
      {
         if ( cadena.startsWith( "em" ) )
            System.out.printf( "\"%s\" empieza con \"em\"\n", cadena );
      } // fin de for

      System.out.println();

      // prueba el método startsWith empezando desde la posición 2 de la cadena
      for ( String cadena: cadenas )
```

Figura 30.4 | Métodos `startsWith` y `endsWith` de la clase `String`. (Parte 1 de 2).

```
21         {
22            if ( cadena.startsWith( "pez", 2 ) )
23               System.out.printf(
24                  "\"%s\" empieza con \"pez\" en la posicion 2\n", cadena );
25         } // fin de for
26
27         System.out.println();
28
29         // prueba el método endsWith
30         for ( String cadena: cadenas )
31         {
32            if ( cadena.endsWith( "do" ) )
33               System.out.printf( "\"%s\" termina con \"do\"\n", cadena );
34         } // fin de for
35      } // fin de main
36   } // fin de la clase CadenaInicioFin
```

```
"empezo" empieza con "em"
"empezando" empieza con "em"

"empezo" empieza con "pez" en la posicion 2
"empezando" empieza con "pez" en la posicion 2

"empezando" termina con "do"
"terminando" termina con "do"
```

Figura 30.4 | Métodos `startsWith` y `endsWith` de la clase `String`. (Parte 2 de 2).

En las líneas 11 a 15 se utiliza la versión del método `startsWith` que recibe un argumento `String`. La condición en la instrucción `if` (línea 13) determina si cada objeto `String` en el arreglo empieza con los caracteres `"em"`. De ser así, el método devuelve `true` y la aplicación imprime ese objeto `String`. En caso contrario, el método devuelve `false` y no ocurre nada.

En las líneas 20 a 25 se utiliza el método `startsWith` que recibe un objeto `String` y un entero como argumentos. El argumento entero especifica el índice en el que debe empezar la comparación en la cadena. La condición en la instrucción `if` (línea 22) determina si cada objeto `String` en el arreglo tiene los caracteres `"pez"`, empezando con el tercer carácter en cada cadena. De ser así, el método devuelve `true` y la aplicación imprime el objeto `String`.

La tercera instrucción `for` (líneas 30 a 34) utiliza el método `endsWith` que recibe un argumento `String`. La condición en la línea 32 determina si cada objeto `String` en el arreglo termina con los caracteres `"do"`. De ser así, el método devuelve `true` y la aplicación imprime el objeto `String`.

30.3.4 Localización de caracteres y subcadenas en las cadenas

A menudo es útil buscar un carácter o conjunto de caracteres en una cadena. Por ejemplo, si usted va a crear su propio procesador de palabras, tal vez quiera proporcionar una herramienta para buscar a través de los documentos. En la figura 30.5 se muestran las diversas versiones de los métodos **`indexOf`** y **`lastIndexOf`** de `String`, que buscan un carácter o subcadena especificados en una cadena. Todas las búsquedas en este ejemplo se realizan en la cadena `letras` (inicializada con `"abcdefghijklmabcdefghijklm"`) que se encuentra en el método `main`. En las líneas 11 a 16 se utiliza el método `indexOf` para localizar la primera ocurrencia de un carácter en una cadena. Si `indexOf` encuentra el carácter, devuelve el índice de ese carácter en la cadena; en caso contrario `indexOf` devuelve `-1`. Hay dos versiones de `indexOf` que buscan caracteres en una cadena. La expresión en la línea 12 utiliza la versión de `indexOf` que recibe una representación entera del carácter que se va a buscar. La expresión en la línea 14 utiliza otra versión del método `indexOf`, que recibe dos argumentos enteros: el carácter y el índice inicial en el que debe empezar la búsqueda en la cadena.

Las instrucciones en las líneas 19 a 24 utilizan el método `lastIndexOf` para localizar la última ocurrencia de un carácter en una cadena. El método `lastIndexOf` realiza la búsqueda desde el final de la cadena, hacia el inicio

de la misma. Si el método `lastIndexOf` encuentra el carácter, devuelve el índice de ese carácter en la cadena; en caso contrario, devuelve –1. Hay dos versiones de `lastIndexOf` que buscan caracteres en una cadena. La expresión en la línea 20 utiliza la versión que recibe la representación entera del carácter. La expresión en la línea 22 utiliza la versión que recibe dos argumentos enteros: la representación entera de un carácter y el índice a partir del cual debe iniciarse la búsqueda inversa de ese carácter.

En las líneas 27 a 40 se muestran las versiones de los métodos `indexOf` y `lastIndexOf` que reciben, cada una de ellas, un objeto `String` como el primer argumento. Estas versiones de los métodos se ejecutan en forma idéntica a las descritas anteriormente, excepto que buscan secuencias de caracteres (o subcadenas) que se especifican mediante sus argumentos `String`. Si se encuentra la subcadena, estos métodos devuelven el índice en la cadena del primer carácter en la subcadena.

```
1  // Fig. 30.5: MetodosIndexString.java
2  // Métodos indexOf y lastIndexOf para buscar en cadenas.
3
4  public class MetodosIndexString
5  {
6     public static void main( String args[] )
7     {
8        String letras = "abcdefghijklmabcdefghijklm";
9
10       // prueba indexOf para localizar un carácter en una cadena
11       System.out.printf(
12          "'c' se encuentra en el indice %d\n", letras.indexOf( 'c' ) );
13       System.out.printf(
14          "'a' se encuentra en el indice %d\n", letras.indexOf( 'a', 1 ) );
15       System.out.printf(
16          "'$' se encuentra en el indice %d\n\n", letras.indexOf( '$' ) );
17
18       // prueba lastIndexOf para buscar un carácter en una cadena
19       System.out.printf( "La ultima 'c' se encuentra en el indice %d\n",
20          letras.lastIndexOf( 'c' ) );
21       System.out.printf( "La ultima 'a' se encuentra en el indice %d\n",
22          letras.lastIndexOf( 'a', 25 ) );
23       System.out.printf( "La ultima '$' se encuentra en el indice %d\n\n",
24          letras.lastIndexOf( '$' ) );
25
26       // prueba indexOf para localizar una subcadena en una cadena
27       System.out.printf( "\"def\" se encuentra en el indice %d\n",
28          letras.indexOf( "def" ) );
29       System.out.printf( "\"def\" se encuentra en el indice %d\n",
30          letras.indexOf( "def", 7 ) );
31       System.out.printf( "\"hola\" se encuentra en el indice %d\n\n",
32          letras.indexOf( "hola" ) );
33
34       // prueba lastIndexOf para buscar una subcadena en una cadena
35       System.out.printf( "La ultima ocurrencia de \"def\" se encuentra en el indice
                            %d\n",
36          letras.lastIndexOf( "def" ) );
37       System.out.printf( "La ultima ocurrencia de \"def\" se encuentra en el indice
                            %d\n",
38          letras.lastIndexOf( "def", 25 ) );
39       System.out.printf( "La ultima ocurrencia de \"hola\" se encuentra en el indice
                            %d\n",
40          letras.lastIndexOf( "hola" ) );
41    } // fin de main
42 } // fin de la clase MetodosIndexString
```

Figura 30.5 | Métodos de búsqueda de la clase `String`. (Parte 1 de 2).

```
'c' se encuentra en el indice 2
'a' se encuentra en el indice 13
'$' se encuentra en el indice -1

La ultima 'c' se encuentra en el indice 15
La ultima 'a' se encuentra en el indice 13
La ultima '$' se encuentra en el indice -1

"def" se encuentra en el indice 3
"def" se encuentra en el indice 16
"hola" se encuentra en el indice -1

La ultima ocurrencia de "def" se encuentra en el indice 16
La ultima ocurrencia de "def" se encuentra en el indice 16
La ultima ocurrencia de "hola" se encuentra en el indice -1
```

Figura 30.5 | Métodos de búsqueda de la clase `String`. (Parte 2 de 2).

30.3.5 Extracción de subcadenas de las cadenas

La clase `String` proporciona dos métodos `substring` para permitir la creación de un nuevo objeto `String` al copiar parte de un objeto `String` existente. Cada método devuelve un nuevo objeto `String`. Ambos métodos se muestran en la figura 30.6.

La expresión `letras.substring( 20 )` en la línea 12 utiliza el método `substring` que recibe un argumento entero. Este argumento especifica el índice inicial en la cadena original `letras`, a partir del cual se van a copiar caracteres. La subcadena devuelta contiene una copia de los caracteres, desde el índice inicial hasta el final de la cadena. Si el índice especificado como argumento se encuentra fuera de los límites de la cadena, el programa genera una excepción **`StringIndexOutOfBoundsException`**.

La expresión `letras.substring( 3, 6 )` en la línea 15 utiliza el método `substring` que recibe dos argumentos enteros. El primer argumento especifica el índice inicial a partir del cual se van a copiar caracteres en la cadena original. El segundo argumento especifica el índice que está una posición más allá del último carácter a copiar (es decir, copiar hasta, pero sin incluir a, ese índice en la cadena). La subcadena devuelta contiene copias de los caracteres especificados de la cadena original. Si los argumentos especificados están fuera de los límites de la cadena, el programa genera una excepción `StringIndexOutOfBoundsException`.

```
1  // Fig. 30.6: SubString.java
2  // Métodos substring de la clase String.
3
4  public class SubString
5  {
6     public static void main( String args[] )
7     {
8        String letras = "abcdefghijklmabcdefghijklm";
9
10       // prueba los métodos substring
11       System.out.printf( "La subcadena desde el índice 20 hasta el final es \"%s\"\n",
12          letras.substring( 20 ) );
13       System.out.printf( "%s \"%s\"\n",
14          "La subcadena desde el índice 3 hasta, pero sin incluir al 6 es",
15          letras.substring( 3, 6 ) );
16    } // fin de main
17 } // fin de la clase SubString
```

```
La subcadena desde el índice 20 hasta el final es "hijklm"
La subcadena desde el índice 3 hasta, pero sin incluir al 6 es "def"
```

Figura 30.6 | Métodos `substring` de la clase `String`.

30.3.6 Concatenación de cadenas

El método **concat** (figura 30.7) de `String` concatena dos objetos `String` y devuelve un nuevo objeto `String`, el cual contiene los caracteres de ambos objetos `String` originales. La expresión `s1.concat( s2 )` de la línea 13 forma una cadena, anexando los caracteres de la cadena `s2` a los caracteres de la cadena `s1`. Los objetos `String` originales a los que se refieren `s1` y `s2` no se modifican.

```
1  // Fig. 30.7: ConcatenacionString.java
2  // Método concat de String.
3
4  public class ConcatenacionString
5  {
6     public static void main( String args[] )
7     {
8        String s1 = new String( "Feliz " );
9        String s2 = new String( "Cumpleanios" );
10
11       System.out.printf( "s1 = %s\ns2 = %s\n\n",s1, s2 );
12       System.out.printf(
13          "Resultado de s1.concat( s2 ) = %s\n", s1.concat( s2 ) );
14       System.out.printf( "s1 despues de la concatenacion= %s\n", s1 );
15    } // fin de main
16 } // fin de la clase ConcatenacionString
```

```
s1 = Feliz
s2 = Cumpleanios

Resultado de s1.concat( s2 ) = Feliz Cumpleanios
s1 despues de la concatenacion= Feliz
```

Figura 30.7 | Método `concat` de `String`.

30.3.7 Métodos varios de `String`

La clase `String` proporciona varios métodos que devuelven copias modificadas de cadenas, o que devuelven arreglos de caracteres. Estos métodos se muestran en la aplicación de la figura 30.8.

```
// Fig. 30.8: VariosString2.java
// Métodos replace, toLowerCase, toUpperCase, trim y toCharArray de String.

public class VariosString2
{
   public static void main( String args[] )
   {
      String s1 = new String( "hola" );
      String s2 = new String( "ADIOS" );
      String s3 = new String( "   espacios   " );

      System.out.printf( "s1 = %s\ns2 = %s\ns3 = %s\n\n", s1, s2, s3 );

      // prueba del método replace
      System.out.printf(
         "Remplazar 'l' con 'L' en s1: %s\n\n", s1.replace( 'l', 'L' ) );

      // prueba de toLowerCase y toUpperCase
      System.out.printf( "s1.toUpperCase() = %s\n", s1.toUpperCase() );
```

Figura 30.8 | Métodos `replace`, `toLowerCase`, `toUpperCase`, `trim` y `toCharArray` de `String`. (Parte 1 de 2).

```
20          System.out.printf( "s2.toLowerCase() = %s\n\n", s2.toLowerCase() );
21
22          // prueba del método trim
23          System.out.printf( "s3 despues de trim = \"%s\"\n\n", s3.trim() );
24
25          // prueba del método toCharArray
26          char arregloChar[] = s1.toCharArray();
27          System.out.print( "s1 como arreglo de caracteres = " );
28
29          for ( char caracter : arregloChar )
30             System.out.print( caracter );
31
32          System.out.println();
33       } // fin de main
34    } // fin de la clase VariosString2
```

```
s1 = hola
s2 = ADIOS
s3 =    espacios

Remplazar 'l' con 'L' en s1: hoLa

s1.toUpperCase() = HOLA
s2.toLowerCase() = adios

s3 despues de trim = "espacios"

s1 como arreglo de caracteres = hola
```

Figura 30.8 | Métodos `replace`, `toLowerCase`, `toUpperCase`, `trim` y `toCharArray` de `String`. (Parte 2 de 2).

En la línea 16 se utiliza el método `replace` de `String` para devolver un nuevo objeto `String`, en el que cada ocurrencia del carácter `'l'` en la cadena `s1` se reemplaza con el carácter `'L'`. El método `replace` no modifica la cadena original. Si no hay ocurrencias del primer argumento en la cadena, el método `replace` devuelve la cadena original.

En la línea 19 se utiliza el método `toUpperCase` de `String` para generar un nuevo objeto `String` con letras mayúsculas, cuyas letras minúsculas correspondientes existen en `s1`. El método devuelve un nuevo objeto `String` que contiene la cadena convertida y deja la cadena original sin cambios. Si no hay caracteres qué convertir, el método `toUpperCase` devuelve la cadena original.

En la línea 20 se utiliza el método `toLowerCase` de `String` para devolver un nuevo objeto `String` con letras minúsculas, cuyas letras mayúsculas correspondientes existen en `s2`. La cadena original permanece sin cambios. Si no hay caracteres qué convertir en la cadena original, `toLowerCase` devuelve la cadena original.

En la línea 23 se utiliza el método **`trim`** de `String` para generar un nuevo objeto `String` que elimine todos los caracteres de espacio en blanco que aparecen al principio o al final en la cadena en la que opera `trim`. El método devuelve un nuevo objeto `String` que contiene a la cadena sin espacios en blanco a la izquierda o a la derecha. La cadena original permanece sin cambios.

En la línea 26 se utiliza el método **`toCharArray`** de `String` para crear un nuevo arreglo de caracteres, el cual contiene una copia de los caracteres en la cadena `s1`. En las líneas 29 y 30 se imprime cada `char` en el arreglo.

30.3.8 Método `valueOf` de `String`

Como hemos visto, todo objeto en Java tiene un método `toString` que permite a un programa obtener la representación de cadena del objeto. Desafortunadamente, esta técnica no puede utilizarse con tipos primitivos, ya que éstos no tienen métodos. La clase `String` proporciona métodos `static` que reciben un argumento de cualquier tipo y lo convierten en un objeto `String`. En la figura 30.9 se muestra el uso de los métodos **`valueOf`** de la clase `String`.

```
1  // Fig. 30.9: StringValueOf.java
2  // Métodos valueOf de String.
3
4  public class StringValueOf
5  {
6     public static void main( String args[] )
7     {
8        char arregloChar[] = { 'a', 'b', 'c', 'd', 'e', 'f' };
9        boolean valorBoolean = true;
10       char valorCaracter = 'Z';
11       int valorEntero = 7;
12       long valorLong = 10000000000L; // el sufijo L indica long
13       float valorFloat = 2.5f; // f indica que 2.5 es un float
14       double valorDouble = 33.333; // no hay sufijo, double es el predeterminado
15       Object refObjeto = "hola"; // asigna la cadena a una referencia Object
16
17       System.out.printf(
18          "arreglo de valores char = %s\n", String.valueOf( arregloChar ) );
19       System.out.printf( "parte del arreglo char = %s\n",
20          String.valueOf( arregloChar, 3, 3 ) );
21       System.out.printf(
22          "boolean = %s\n", String.valueOf( valorBoolean ) );
23       System.out.printf(
24          "char = %s\n", String.valueOf( valorCaracter ) );
25       System.out.printf( "int = %s\n", String.valueOf( valorEntero ) );
26       System.out.printf( "long = %s\n", String.valueOf( valorLong ) );
27       System.out.printf( "float = %s\n", String.valueOf( valorFloat ) );
28       System.out.printf(
29          "double = %s\n", String.valueOf( valorDouble ) );
30       System.out.printf( "Object = %s\n", String.valueOf( refObjeto ) );
31    } // fin de main
32 } // fin de la clase StringValueOf
```

```
arreglo de valores char = abcdef
parte del arreglo char = def
boolean = true
char = Z
int = 7
long = 10000000000
float = 2.5
double = 33.333
Object = hola
```

Figura 30.9 | Métodos `valueOf` de la clase `String`.

La expresión `String.valueOf( arregloChar )` en la línea 18 utiliza el arreglo de caracteres `arregloChar` para crear un nuevo objeto `String`. La expresión `String.valueOf( arregloChar, 3, 3 )` de la línea 20 utiliza una porción del arreglo de caracteres `arregloChar` para crear un nuevo objeto `String`. El segundo argumento especifica el índice inicial a partir del cual se van a utilizar caracteres. El tercer argumento especifica el número de caracteres a usar.

Existen otras siete versiones del método `valueOf`, las cuales toman argumentos de tipo `boolean`, `char`, `int`, `long`, `float`, `double` y `Object`, respectivamente. Estas versiones se muestran en las líneas 21 a 30. Observe que la versión de `valueOf` que recibe un objeto `Object` como argumento puede hacerlo debido a que todos los objetos `Object` pueden convertirse en objetos `String` mediante el método `toString`.

[*Nota:* en las líneas 12 y 13 se utilizan los valores literales `10000000000L` y `2.5f` como valores iniciales de la variable `valorLong` tipo `long` y de la variable `valorFloat` tipo `float`, respectivamente. De manera predeterminada, Java trata a las literales enteras como tipo `int` y a las literales de punto flotante como tipo `double`. Si se anexa la letra `L` a la literal `10000000000` y se anexa la letra `f` a la literal `2.5`, se indica al compilador que `10000000000`

debe tratarse como `long` y que `2.5` debe tratarse como `float`. Se puede usar una `L` mayúscula o una `l` minúscula para denotar una variable de tipo `long`, y una `F` mayúscula o una `f` minúscula para denotar una variable de tipo `float`].

30.4 La clase StringBuilder

Una vez que se crea un objeto `String`, su contenido ya no puede variar. Ahora hablaremos sobre las características de la clase **`StringBuilder`** para crear y manipular información de cadenas dinámicas; es decir, cadenas que pueden modificarse. Cada objeto `StringBuilder` es capaz de almacenar varios caracteres especificados por su capacidad. Si se excede la capacidad de un objeto `StringBuilder`, ésta se expande de manera automática para dar cabida a los caracteres adicionales. La clase `StringBuilder` también se utiliza para implementar los operadores + y += para la concatenación de objetos `String`.



Tip de rendimiento 30.2

Java puede realizar ciertas optimizaciones en las que se involucran objetos `String` (como compartir un objeto `String` entre varias referencias), ya que sabe que estos objetos no cambiarán. Si los datos no van a cambiar, debemos usar objetos `String` (y no `StringBuilder`).



Tip de rendimiento 30.3

En los programas que realizan la concatenación de cadenas con frecuencia, o en otras modificaciones de cadenas, a menudo es más eficiente implementar las modificaciones con la clase `StringBuilder`.



Observación de ingeniería de software 30.2

*Los objetos `StringBuilder` no son seguros para los subprocesos. Si varios subprocesos requieren acceso a la misma información de una cadena dinámica, use la clase **`StringBuffer`** en su código. Las clases `StringBuilder` y `StringBuffer` son idénticas, pero la clase `StringBuffer` es segura para los subprocesos.*

30.4.1 Constructores de StringBuilder

La clase `StringBuilder` proporciona cuatro constructores. En la figura 30.10 mostramos tres de ellos. En la línea 8 se utiliza el constructor de `StringBuilder` sin argumentos para crear un objeto `StringBuilder` que no contiene caracteres, y tiene una capacidad inicial de 16 caracteres (el valor predeterminado para un objeto `StringBuilder`). En la línea 9 se utiliza el constructor de `StringBuilder` que recibe un argumento entero para crear un objeto `StringBuilder` que no contiene caracteres, y su capacidad inicial se especifica mediante el argumento entero (es decir, `10`). En la línea 10 se utiliza el constructor de `StringBuilder` que recibe un argumento `String` (en este caso, una literal de cadena) para crear un objeto `StringBuilder` que contiene los caracteres en el argumento `String`. La capacidad inicial es el número de caracteres en el argumento `String`, más 16.

Las instrucciones en las líneas 12 a 14 utilizan el método `toString` de la clase `StringBuilder` para imprimir los objetos `StringBuilder` con el método `printf`. En la sección 30.4.4, hablaremos acerca de cómo Java usa los objetos `StringBuilder` para implementar los operadores + y += para la concatenación de cadenas.

```
1  // Fig. 30.10: ConstructoresStringBuilder.java
2  // Constructores de StringBuilder.
3
4  public class ConstructoresStringBuilder
5  {
6     public static void main( String args[] )
7     {
8        StringBuilder bufer1 = new StringBuilder();
9        StringBuilder bufer2 = new StringBuilder( 10 );
10       StringBuilder bufer3 = new StringBuilder( "hola" );
11
```

Figura 30.10 | Constructores de la clase `StringBuilder`. (Parte 1 de 2).

```
      System.out.printf( "bufer1 = \"%s\"\n", bufer1.toString() );
      System.out.printf( "bufer2 = \"%s\"\n", bufer2.toString() );
      System.out.printf( "bufer3 = \"%s\"\n", bufer3.toString() );
   } // fin de main
} // fin de la clase ConstructoresStringBuilder
```

```
bufer1 = ""
bufer2 = ""
bufer3 = "hola"
```

Figura 30.10 | Constructores de la clase `StringBuilder`. (Parte 2 de 2).

30.4.2 Métodos `length`, `capacity`, `setLength` y `ensureCapacity` de `StringBuilder`

La clase `StringBuilder` proporciona los métodos `length` y `capacity` para devolver el número actual de caracteres en un objeto `StringBuilder` y el número de caracteres que pueden almacenarse en un objeto `StringBuilder` sin necesidad de asignar más memoria, respectivamente. El método `ensureCapacity` permite al programador garantizar que un `StringBuilder` tenga, cuando menos, la capacidad especificada. El método `setLength` permite al programador incrementar o decrementar la longitud de un objeto `StringBuilder`. En la figura 30.11 se muestra el uso de estos métodos.

La aplicación contiene un objeto `StringBuilder` llamado `bufer`. En la línea 8 se utiliza el constructor de `StringBuilder` que toma un argumento `String` para inicializar el objeto `StringBuilder` con la cadena `"Hola, como estas?"`. En las líneas 10 y 11 se imprimen el contenido, la longitud y la capacidad del objeto `StringBuilder`. Observe en la ventana de salida que la capacidad del objeto `StringBuilder` es inicialmente de 35. Recuerde que el constructor de `StringBuilder` que recibe un argumento `String` inicializa la capacidad con la longitud de la cadena que se le pasa como argumento, más 16.

En la línea 13 se utiliza el método `ensureCapacity` para expandir la capacidad del objeto `StringBuilder` a un mínimo de 75 caracteres. En realidad, si la capacidad original es menor que el argumento, este método asegura una capacidad que sea el valor mayor entre el número especificado como argumento, y el doble de la capacidad original más 2. Si la capacidad actual del objeto `StringBuilder` es más que la capacidad especificada, la capacidad del objeto `StringBuilder` permanece sin cambios.

```
1  // Fig. 30.11: StringBuilderCapLen.java
2  // Métodos length, setLength, capacity y ensureCapacity de StringBuilder.
3
4  public class StringBuilderCapLen
5  {
6     public static void main( String args[] )
7     {
8        StringBuilder bufer = new StringBuilder( "Hola, como estas?" );
9
10       System.out.printf( "bufer = %s\nlongitud = %d\ncapacidad = %d\n\n",
11          bufer.toString(), bufer.length(), bufer.capacity() );
12
13       bufer.ensureCapacity( 75 );
14       System.out.printf( "Nueva capacidad = %d\n\n", bufer.capacity() );
15
16       bufer.setLength( 10 );
17       System.out.printf( "Nueva longitud = %d\nbufer = %s\n",
18          bufer.length(), bufer.toString() );
19    } // fin de main
20 } // fin de la clase StringBuilderCapLen
```

Figura 30.11 | Métodos `length` y `capacity` de `StringBuilder`. (Parte 1 de 2).

```
bufer = Hola, como estas?
longitud = 17
capacidad = 33

Nueva capacidad = 75

Nueva longitud = 10
bufer = Hola, como
```

Figura 30.11 | Métodos `length` y `capacity` de `StringBuilder`. (Parte 2 de 2).



Tip de rendimiento 30.4

El proceso de incrementar en forma dinámica la capacidad de un objeto `StringBuilder` *puede requerir una cantidad considerable de tiempo. La ejecución de un gran número de estas operaciones puede degradar el rendimiento de una aplicación. Si un objeto* `StringBuilder` *va a aumentar su tamaño en forma considerable, tal vez varias veces, si establecemos su capacidad a un nivel alto en un principio se incrementará el rendimiento.*

En la línea 16 se utiliza el método `setLength` para establecer la longitud del objeto `StringBuilder` en 10. Si la longitud especificada es menor que el número actual de caracteres en el objeto `StringBuilder`, el búfer se trunca a la longitud especificada (es decir, los caracteres en el objeto `StringBuilder` después de la longitud especificada se descartan). Si la longitud especificada es mayor que el número de caracteres actualmente en el objeto `StringBuilder`, se anexan caracteres `nulos` (caracteres con la representación numérica de 0) al objeto `StringBuilder` hasta que el número total de caracteres en el objeto `StringBuilder` sea igual a la longitud especificada.

30.4.3 Métodos charAt, setCharAt, getChars y reverse de StringBuilder

La clase `StringBuilder` proporciona los métodos **`charAt`**, **`setCharAt`**, **`getChars`** y **`reverse`** para manipular los caracteres en un objeto StringBuffer. Cada uno de estos métodos se muestra en la figura 30.12.

```java
// Fig. 30.12: StringBuilderChars.java
// Métodos charAt, setCharAt, getChars y reverse de StringBuilder.

public class StringBuilderChars
{
   public static void main( String args[] )
   {
      StringBuilder bufer = new StringBuilder( "hola a todos" );

      System.out.printf( "bufer = %s\n", bufer.toString() );
       System.out.printf( "Caracter en 0: %s\nCaracter en 3: %s\n\n",
         bufer.charAt( 0 ), bufer.charAt( 3 ) );

      char arregloChars[] = new char[ bufer.length() ];
      bufer.getChars( 0, bufer.length(), arregloChars, 0 );
      System.out.print( "Los caracteres son: " );

      for ( char character : arregloChars )
         System.out.print( character );

      bufer.setCharAt( 0, 'H' );
      bufer.setCharAt( 7, 'T' );
      System.out.printf( "\n\nbufer = %s", bufer.toString() );
```

Figura 30.12 | Métodos para la manipulación de caracteres de `StringBuilder`. (Parte 1 de 2).

```
24
25          bufer.reverse();
26          System.out.printf( "\n\nbufer = %s\n", bufer.toString() );
27       } // fin de main
28    } // fin de la clase StringBuilderChars
```

```
bufer  = hola a todos
Caracter en 0: h
Caracter en 3: a

Los caracteres son: hola a todos

bufer  = Hola a Todos

bufer  = sodoT a aloH
```

Figura 30.12 | Métodos para la manipulación de caracteres de `StringBuilder`. (Parte 2 de 2).

El método `charAt` (línea 12) recibe un argumento entero y devuelve el carácter que se encuentre en el objeto `StringBuilder` en ese índice. El método `getChars` (línea 15) copia los caracteres de un objeto `StringBuilder` al arreglo de caracteres que recibe como argumento. Este método recibe cuatro argumentos: el índice inicial a partir del cual deben copiarse caracteres en el objeto `StringBuilder`, el índice una posición más allá del último carácter a copiar del objeto `StringBuilder`, el arreglo de caracteres en el que se van a copiar los caracteres y la posición inicial en el arreglo de caracteres en donde debe colocarse el primer carácter. El método `setCharAt` (líneas 21 y 22) recibe un entero y un carácter como argumentos y asigna al carácter en la posición especificada en el objeto `StringBuilder` el carácter que recibe como argumento. El método `reverse` (línea 25) invierte el contenido del objeto `StringBuilder`.



Error común de programación 30.3

Al tratar de acceder a un carácter que se encuentra fuera de los límites de un objeto `StringBuilder` (es decir, con un índice menor que 0, o mayor o igual que la longitud del objeto `StringBuilder`) se produce una excepción `StringIndexOutOfBoundsException`.

30.4.4 Métodos append de StringBuilder

La clase `StringBuilder` proporciona métodos **append** sobrecargados (que mostramos en la figura 30.13) para permitir que se agreguen valores de diversos tipos al final de un objeto `StringBuilder`. Se proporcionan versiones para cada uno de los tipos primitivos y para arreglos de caracteres, objetos `String`, `Object`, `StringBuilder` y `CharSequence` (recuerde que el método `toString` produce una representación de cadena de cualquier objeto `Object`). Cada uno de los métodos recibe su argumento, lo convierte en una cadena y lo anexa al objeto `StringBuilder`.

```
// Fig. 30.13: StringBuilderAppend.java
// Métodos append de StringBuilder.

public class StringBuilderAppend
{
   public static void main( String args[] )
   {
      Object refObjeto = "hola";
      String cadena = "adios";
      char arregloChar[] = { 'a', 'b', 'c', 'd', 'e', 'f' };
      boolean valorBoolean = true;
```

Figura 30.13 | Métodos `append` de la clase `StringBuilder`. (Parte 1 de 2).

```
        char valorChar = 'Z';
        int valorInt = 7;
        long valorLong = 10000000000L;
        float valorFloat = 2.5f; // el sufijo f indica que 2.5 es un float
        double valorDouble = 33.333;

        StringBuilder ultimoBufer = new StringBuilder( "ultimo bufer" );
        StringBuilder bufer = new StringBuilder();

        bufer.append( refObjeto );
        bufer.append( "\n" ); // cada uno de estos contiene nueva línea
        bufer.append( cadena );
        bufer.append( "\n" );
        bufer.append( arregloChar );
        bufer.append( "\n" );
        bufer.append( arregloChar, 0, 3 );
        bufer.append( "\n" );
        bufer.append( valorBoolean );
        bufer.append( "\n" );
        bufer.append( valorChar );
        bufer.append( "\n" );
        bufer.append( valorInt );
        bufer.append( "\n" );
        bufer.append( valorLong );
        bufer.append( "\n" );
        bufer.append( valorFloat );
        bufer.append( "\n" );
        bufer.append( valorDouble );
        bufer.append( "\n" );
        bufer.append( ultimoBufer );

        System.out.printf( "bufer contiene %s\n", bufer.toString() );
    } // fin de main
} // fin de StringBuilderAppend
```

```
bufer contiene hola
adios
abcdef
abc
true
Z
7
10000000000
2.5
33.333
ultimo bufer
```

Figura 30.13 | Métodos append de la clase StringBuilder. (Parte 2 de 2).

En realidad, el compilador utiliza los objetos StringBuilder y los métodos append para implementar los operadores + y += para concatenar objetos String. Por ejemplo, suponga que se realizan las siguientes declaraciones:

```
String cadena1 = "hola";
String cadena2 = "BC"
int valor = 22;
```

la instrucción

```
String s = cadena1 + cadena2 + valor;
```

concatena a "hola", "BC" y 22. La concatenación se realiza de la siguiente manera:

```
new StringBuilder().append( "hola" ).append( "BC" ).append( 22 ).toString();
```

Primero, Java crea un objeto `StringBuilder` vacío y después anexa a este objeto `StringBuffer` las cadenas "hola", "BC" y el entero 22. A continuación, el método `toString` de `StringBuilder` convierte el objeto `StringBuilder` en un objeto `String` que se asigna al objeto `String` `s`. La instrucción

```
s += "!";
```

se ejecuta de la siguiente manera:

```
s = new StringBuilder().append( s ).append( "!" ).toString();
```

Primero, Java crea un objeto `StringBuilder` vacío y después anexa a ese objeto `StringBuilder` el contenido actual de `s`, seguido por "!". A continuación, el método `toString` de `StringBuilder` convierte el objeto `StringBuilder` en una representación de cadena, y el resultado se asigna a `s`.

30.4.5 Métodos de inserción y eliminación de StringBuilder

La clase `StringBuilder` proporciona métodos **`insert`** sobrecargados para permitir que se inserten valores de diversos tipos en cualquier posición de un objeto `StringBuilder`. Se proporcionan versiones para cada uno de los tipos primitivos, y para arreglos de caracteres, objetos `String`, `Object` y `CharSequence`. Cada uno de los métodos toma su segundo argumento, lo convierte en una cadena y la inserta justo antes del índice especificado por el primer argumento. El primer argumento debe ser mayor o igual que 0, y menor que la longitud del objeto `StringBuilder`; de no ser así, se produce una excepción `StringIndexOutOfBoundsException`. La clase `StringBuilder` también proporciona métodos **`delete`** y **`deleteCharAt`** para eliminar caracteres en cualquier posición de un objeto `StringBuilder`. El método `delete` recibe dos argumentos: el índice inicial y el índice que se encuentra una posición más allá del último de los caracteres que se van a eliminar. Se eliminan todos los caracteres que empiezan en el índice inicial hasta, pero sin incluir al índice final. El método `deleteCharAt` recibe un argumento: el índice del carácter a eliminar. El uso de índices inválidos hace que ambos métodos lancen una excepción `StringIndexOutOfBoundsException`. Los métodos `insert`, `delete` y `deleteCharAt` se muestran en la figura 30.14.

```
// Fig. 30.14: StringBuilderInsert.java
// Métodos insert, delete y deleteCharAt de StringBuilder

public class StringBuilderInsert
{
   public static void main( String args[] )
   {
      Object refObjeto = "hola";
      String cadena = "adios";
      char arregloChars[] = { 'a', 'b', 'c', 'd', 'e', 'f' };
      boolean valorBoolean = true;
      char valorChar = 'K';
      int valorInt = 7;
      long valorLong = 10000000;
      float valorFloat = 2.5f; // el sufijo f indica que 2.5 es un float
      double valorDouble = 33.333;

      StringBuilder bufer = new StringBuilder();

      bufer.insert( 0, refObjeto );
      bufer.insert( 0, "  " ); // cada uno de estos contiene nueva línea
      bufer.insert( 0, cadena );
      bufer.insert( 0, "  " );
```

Figura 30.14 | Métodos `insert` y `delete` de `StringBuilder`. (Parte 1 de 2).

```
        bufer.insert( 0, arregloChars );
        bufer.insert( 0, "  " );
        bufer.insert( 0, arregloChars, 3, 3 );
        bufer.insert( 0, "  " );
        bufer.insert( 0, valorBoolean );
        bufer.insert( 0, "  " );
        bufer.insert( 0, valorChar );
        bufer.insert( 0, "  " );
        bufer.insert( 0, valorInt );
        bufer.insert( 0, "  " );
        bufer.insert( 0, valorLong );
        bufer.insert( 0, "  " );
        bufer.insert( 0, valorFloat );
        bufer.insert( 0, "  " );
        bufer.insert( 0, valorDouble );

        System.out.printf(
           "bufer despues de insertar:\n%s\n\n", bufer.toString() );

        bufer.deleteCharAt( 10 ); // elimina el 5 en 2.5
        bufer.delete( 2, 6 ); // elimina el .333 en 33.333

        System.out.printf(
           "bufer despues de eliminar:\n%s\n", bufer.toString() );
     } // fin de main
  } // fin de la clase StringBuilderInsert
```

```
bufer despues de insertar:
33.333  2.5  10000000  7  K  true  def  abcdef  adios  hola

bufer despues de eliminar:
33  2.  10000000  7  K  true  def  abcdef  adios  hola
```

Figura 30.14 | Métodos `insert` y `delete` de `StringBuilder`. (Parte 2 de 2).

30.5 La clase Character

En el capítulo 17 vimos que Java proporciona ocho clases de envoltura de tipos: `Boolean`, `Character`, `Double`, `Float`, `Byte`, `Short`, `Integer` y `Long`, los cuales permiten que los valores de tipo primitivo sean tratados como objetos. En esta sección presentaremos la clase `Character`: la clase de envoltura de tipos para el tipo primitivo `char`.

La mayoría de los métodos de la clase `Character` son `static`, diseñados para facilitar el procesamiento de valores `char` individuales. Estos métodos reciben cuando menos un argumento tipo carácter y realizan una prueba o manipulación del carácter. Esta clase también contiene un constructor que recibe un argumento `char` para inicializar un objeto `Character`. En los siguientes tres ejemplos presentaremos la mayor parte de los métodos de la clase `Character`. Para obtener más información sobre esta clase (y las demás clases de envoltura de tipos), consulte el paquete `java.lang` en la documentación del API de Java.

En la figura 30.15 se muestran algunos métodos `static` que prueban caracteres para determinar si son de un tipo de carácter específico y los métodos `static` que realizan conversiones de caracteres de minúscula a mayúscula, y viceversa. Puede introducir cualquier carácter y aplicar estos métodos a ese carácter.

En la línea 15 se utiliza el método **isDefined** de `Character` para determinar si el carácter `c` está definido en el conjunto de caracteres Unicode. De ser así, el método devuelve `true`; en caso contrario, devuelve `false`. En la línea 16 se utiliza el método **isDigit** de `Character` para determinar si el carácter `c` es un dígito definido en Unicode. De ser así el método devuelve `true` y, en caso contrario devuelve `false`.

En la línea 18 se utiliza el método **isJavaIdentifierStart** de `Character` para determinar si `c` es un carácter que puede ser el primer carácter de un identificador en Java; es decir, una letra, un guión bajo (_) o un signo de dólares ($). De ser así, el método devuelve `true`; en caso contrario devuelve `false`. En la línea 20 se utiliza

```java
// Fig. 30.15: MetodosStaticChar.java
// Prueba de los métodos static de Character y los métodos de conversión de mayúsculas/minúsculas.
import java.util.Scanner;

public class MetodosStaticChar
{
   public static void main( String args[] )
   {
      Scanner scanner = new Scanner( System.in ); // crea objeto scanner
      System.out.println( "Escriba un caracter y oprima Intro" );
      String entrada = scanner.next();
      char c = entrada.charAt( 0 ); // obtiene el caracter de entrada

      // muestra información sobre los caracteres
      System.out.printf( "esta definido: %b\n", Character.isDefined( c ) );
      System.out.printf( "es digito: %b\n", Character.isDigit( c ) );
      System.out.printf( "es el primer caracter en un identificador de Java: %b\n",
         Character.isJavaIdentifierStart( c ) );
      System.out.printf( "es parte de un identificador de Java: %b\n",
         Character.isJavaIdentifierPart( c ) );
      System.out.printf( "es letra: %b\n", Character.isLetter( c ) );
      System.out.printf(
         "es letra o digito: %b\n", Character.isLetterOrDigit( c ) );
      System.out.printf(
         "es minuscula: %b\n", Character.isLowerCase( c ) );
      System.out.printf(
         "es mayuscula: %b\n", Character.isUpperCase( c ) );
      System.out.printf(
         "a mayuscula: %s\n", Character.toUpperCase( c ) );
      System.out.printf(
         "a minuscula: %s\n", Character.toLowerCase( c ) );
   } // fin de main
} // fin de la clase MetodosStaticChar
```

```
Escriba un caracter y oprima Intro
A
esta definido: true
es digito: false
es el primer caracter en un identificador de Java: true
es parte de un identificador de Java: true
es letra: true
es letra o digito: true
es minuscula: false
es mayuscula: true
a mayuscula: A
a minuscula: a
```

```
Escriba un caracter y oprima Intro
8
esta definido: true
es digito: true
es el primer caracter en un identificador de Java: false
es parte de un identificador de Java: true
es letra: false
es letra o digito: true
```

Figura 30.15 | Métodos `static` de la clase `Character` para probar caracteres y convertir de mayúsculas a minúsculas, y viceversa. (Parte 1 de 2).

```
es minuscula: false
es mayuscula: false
a mayuscula: 8
a minuscula: 8
```

```
Escriba un caracter y oprima Intro
$
esta definido: true
es digito: false
es el primer caracter en un identificador de Java: true
es parte de un identificador de Java: true
es letra: false
es letra o digito: false
es minuscula: false
es mayuscula: false
a mayuscula: $
a minuscula: $
```

Figura 30.15 | Métodos `static` de la clase `Character` para probar caracteres y convertir de mayúsculas a minúsculas, y viceversa. (Parte 2 de 2).

el método **`isJavaIdentifierPart`** de `Character` para determinar si el carácter `c` puede utilizarse en un identificador en Java; es decir, un dígito, una letra, un guión bajo (_) o un signo de dólares (`$`). De ser así, el método devuelve `true`; en caso contrario devuelve `false`.

En la línea 21 se utiliza el método **`isLetter`** de `Character` para determinar si el carácter `c` es una letra. Si es así, el método devuelve `true`; en caso contrario devuelve `false`. En la línea 23 se utiliza el método **`isLetterOrDigit`** de `Character` para determinar si el carácter `c` es una letra o un dígito. Si es así, el método devuelve `true`; en caso contrario devuelve `false`.

En la línea 25 se utiliza el método **`isLowerCase`** de `Character` para determinar si el carácter `c` es una letra minúscula. Si es así, el método devuelve `true`; en caso contrario devuelve `false`. En la línea 27 se utiliza el método **`isUpperCase`** de `Character` para determinar si el carácter `c` es una letra mayúscula. Si es así, el método devuelve `true`; en caso contrario devuelve `false`.

En la línea 29 se utiliza el método **`toUpperCase`** de `Character` para convertir el carácter `c` en su letra mayúscula equivalente. El método devuelve el carácter convertido si éste tiene un equivalente en mayúscula; en caso contrario, el método devuelve su argumento original. En la línea 31 se utiliza el método `toLowerCase` de `Character` para convertir el carácter `c` en su letra minúscula equivalente. El método devuelve el carácter convertido si éste tiene un equivalente en minúscula; en caso contrario, el método devuelve su argumento original.

En la figura 30.16 se muestran los métodos estáticos **`digit`** y **`forDigit`** de `Character`, los cuales convierten caracteres a dígitos y dígitos a caracteres, respectivamente, en distintos sistemas numéricos. Los sistemas numéricos comunes son el decimal (base 10), octal (base 8), hexadecimal (base 16) y binario (base 2). La base de un número se conoce también como su **raíz**. Para obtener más información sobre las conversiones entre sistemas numéricos, vea el apéndice E.

```
// Fig. 30.16: MetodosStaticChar2.java
// Métodos de conversión estáticos de Character.
import java.util.Scanner;

public class MetodosStaticChar2
{
   // crea el objeto MetodosStaticChar2 y ejecuta la aplicación
   public static void main( String args[] )
   {
      Scanner scanner = new Scanner( System.in );
```

Figura 30.16 | Métodos de conversión `static` de la clase `Character`. (Parte 1 de 2).

```
11
12          // obtiene la raíz
13          System.out.println( "Escriba una raiz:" );
14          int raiz = scanner.nextInt();
15
16          // obtiene la selección del usuario
17          System.out.printf( "Seleccione una opcion:\n1 -- %s\n2 -- %s\n",
18             "Convertir digito a caracter", "Convertir caracter a digito" );
19          int opcion = scanner.nextInt();
20
21          // procesa la petición
22          switch ( opcion )
23          {
24             case 1: // convierte dígito a carácter
25                System.out.println( "Escriba un digito:" );
26                int digito = scanner.nextInt();
27                System.out.printf( "Convertir digito a caracter: %s\n",
28                   Character.forDigit( digito, raiz ) );
29                break;
30
31             case 2: // convierte carácter a dígito
32                System.out.println( "Escriba un caracter:" );
33                char caracter = scanner.next().charAt( 0 );
34                System.out.printf( "Convertir caracter a digito: %s\n",
35                   Character.digit( caracter, raiz ) );
36                break;
37          } // fin de switch
38       } // fin de main
39    } // fin de la clase MetodosStaticChar2
```

```
Escriba una raiz:
16
Seleccione una opcion:
1 -- Convertir digito a caracter
2 -- Convertir caracter a digito
2
Escriba un caracter:
A
Convertir caracter a digito: 10
```

```
Escriba una raiz:
16
Seleccione una opcion:
1 -- Convertir digito a caracter
2 -- Convertir caracter a digito
1
Escriba un digito:
13
Convertir digito a caracter: d
```

Figura 30.16 | Métodos de conversión `static` de la clase `Character`. (Parte 2 de 2).

En la línea 28 se utiliza el método `forDigit` para convertir el entero `digito` en un carácter del sistema numérico especificado por el entero `raiz` (la base del número). Por ejemplo, el entero decimal `13` en base 16 (la `raiz`) tiene el valor de carácter `'d'`. Observe que las letras en minúsculas y mayúsculas representan el mismo valor en los sistemas numéricos. En la línea 35 se utiliza el método `digit` para convertir el carácter `c` en un entero del sistema numérico especificado por el entero `raiz` (la base del número). Por ejemplo, el carácter `'A'` es la representación en base 16 (la `raiz`) del valor 10 en base 10. La raíz debe estar entre 2 y 36, inclusive.

En la figura 30.17 se muestra el constructor y varios métodos no static de la clase Character: **charValue**, toString y equals. En las líneas 8 y 9 se instancian dos objetos Character al realizar conversiones autoboxing en las constantes de caracteres 'A' y 'a', respectivamente. En la línea 12 se utiliza el método charValue de Character para devolver el valor char almacenado en el objeto Character llamado c1. En la línea 12 se devuelve la representación de cadena del objeto Character llamado c2, utilizando el método toString. La condición en la instrucción if...else de las líneas 14 a 17 utiliza el método equals para determinar si el objeto c1 tiene el mismo contenido que el objeto c2 (es decir, si los caracteres dentro de cada objeto son iguales).

```
1  // Fig. 30.17: OtrosMetodosChar.java
2  // Métodos no static de Character.
3
4  public class OtrosMetodosChar
5  {
6     public static void main( String args[] )
7     {
8        Character c1 = 'A';
9        Character c2 = 'a';
10
11       System.out.printf(
12          "c1 = %s\nc2 = %s\n\n", c1.charValue(), c2.toString() );
13
14       if ( c1.equals( c2 ) )
15          System.out.println( "c1 y c2 son iguales\n" );
16       else
17          System.out.println( "c1 y c2 no son iguales\n" );
18    } // fin de main
19 } // fin de la clase OtrosMetodosChar
```

```
c1 = A
c2 = a

c1 y c2 no son iguales
```

Figura 30.17 | Métodos no static de la clase Character.

30.6 La clase StringTokenizer

Cuando usted lee una oración, su mente la divide en **tokens** (palabras individuales y signos de puntuación, cada uno de los cuales transfiere a usted su significado). Los compiladores también llevan a cabo la descomposición de instrucciones en piezas individuales tales como palabras clave, identificadores, operadores y demás elementos de un lenguaje de programación. Ahora estudiaremos la clase StringTokenizer de Java (del paquete java.util), la cual descompone una cadena en los tokens que la componen. Los tokens se separan unos de otros mediante **delimitadores**, que generalmente son caracteres de espacio en blanco tales como los espacios, tabuladores, nuevas líneas y retornos de carro. También pueden utilizarse otros caracteres como delimitadores para separar tokens. La aplicación de la figura 30.18 muestra el uso de la clase StringTokenizer.

Cuando el usuario oprime *Intro*, el enunciado de entrada se almacena en la variable enunciado. En la línea 17 se crea un objeto StringTokenizer para enunciado. El constructor de StringTokenizer recibe un argumento de cadena y crea un objeto StringTokenizer para esa cadena, utilizando la cadena delimitadora predeterminada "\t\n\r\f" que consiste en un espacio, un tabulador, un retorno de carro y una nueva línea, para la descomposición en tokens. Hay otros dos constructores para la clase StringTokenizer. En la versión que recibe dos argumentos String, el segundo String es la cadena delimitadora. En la versión que recibe tres argumentos, el segundo String es la cadena delimitadora y el tercer argumento (boolean) determina si los delimitadores también se devuelven como tokens (sólo si este argumento es true). Esto es útil si usted necesita saber cuáles son los delimitadores.

En la línea 19 se utiliza el método countTokens de StringTokenizer para determinar el número de tokens en la cadena que se va a descomponer en tokens. La condición en la línea 21 utiliza el método **hasMoreTokens**

```
1  // Fig. 30.18: PruebaToken.java
2  // La clase StringTokenizer.
3  import java.util.Scanner;
4  import java.util.StringTokenizer;
5
6  public class PruebaToken
7  {
8     // ejecuta la aplicación
9     public static void main( String args[] )
10    {
11       // obtiene el enunciado
12       Scanner scanner = new Scanner( System.in );
13       System.out.println( "Escriba un enunciado y oprima Intro" );
14       String enunciado = scanner.nextLine();
15
16       // procesa el enunciado del usuario
17       StringTokenizer tokens = new StringTokenizer( enunciado );
18       System.out.printf( "Numero de elementos: %d\nLos tokens son:\n",
19          tokens.countTokens() );
20
21       while ( tokens.hasMoreTokens() )
22          System.out.println( tokens.nextToken() );
23    } // fin de main
24 } // fin de la clase PruebaToken
```

```
Escriba un enunciado y oprima Intro
Este es un enunciado con siete tokens
Numero de elementos: 7
Los tokens son:
Este
es
un
enunciado
con
siete
tokens
```

Figura 30.18 | Objeto `StringTokenizer` utilizado para descomponer cadenas en tokens.

de `StringTokenizer` para determinar si hay más tokens en la cadena que va a descomponerse. De ser así, en la línea 22 se imprime el siguiente token en el objeto `String`. El siguiente token se obtiene mediante una llamada al método **nextToken** de `StringTokenizer`, el cual devuelve un objeto `String`. El token se imprime mediante el uso de `println`, de manera que los siguientes tokens aparezcan en líneas separadas.

Si desea cambiar la cadena delimitadora mientras descompone una cadena en tokens, puede hacerlo especificando una nueva cadena delimitadora en una llamada a `nextToken`, como se muestra a continuación:

```
tokens.nextToken( nuevaCadenaDelimitadora );
```

Esta característica no se muestra en la figura 30.18.

30.7 Expresiones regulares, la clase `Pattern` y la clase `Matcher`

Las **expresiones regulares** son secuencias de caracteres y símbolos que definen un conjunto de cadenas. Son útiles para validar la entrada y asegurar que los datos estén en un formato específico. Por ejemplo, un código postal debe consistir de cinco dígitos, y un apellido sólo debe contener letras, espacios, apóstrofes y guiones cortos. Una aplicación de las expresiones regulares es facilitar la construcción de un compilador. A menudo se utiliza una expresión regular larga y compleja para validar la sintaxis de un programa. Si el código del programa no coincide con la expresión regular, el compilador sabe que hay un error de sintaxis dentro del código.

La clase `String` proporciona varios métodos para realizar operaciones con expresiones regulares, siendo la más simple la operación de concordancia. El método **`matches`** de la clase `String` recibe una cadena que especifica la expresión regular, e iguala el contenido del objeto `String` que lo llama con la expresión regular. Este método devuelve un valor de tipo `boolean` indicando si hubo concordancia o no.

Una expresión regular consiste de caracteres literales y símbolos especiales. La tabla de la figura 30.19 especifica algunas **clases predefinidas de caracteres** que pueden usarse con las expresiones regulares. Una clase de carácter es una secuencia de escape que representa a un grupo de caracteres. Un dígito es cualquier carácter numérico. Un **carácter de palabra** es cualquier letra (mayúscula o minúscula), cualquier dígito o el carácter de guión bajo. Un carácter de espacio en blanco es un espacio, tabulador, retorno de carro, nueva línea o avance de página. Cada clase de carácter se iguala con un solo carácter en la cadena que intentamos hacer concordar con la expresión regular.

Las expresiones regulares no están limitadas a esas clases predefinidas de caracteres. Las expresiones utilizan varios operadores y otras formas de notación para igualar patrones complejos. Analizaremos varias de estas técnicas en la aplicación de las figuras 30.20 y 30.21, la cual valida la entrada del usuario mediante expresiones regulares. [*Nota:* esta aplicación no está diseñada para igualar todos los posibles datos de entrada del usuario].

Carácter	Concuerda con	Carácter	Concuerda con
`\d`	cualquier dígito	`\D`	cualquier carácter que no sea dígito
`\w`	cualquier carácter de palabra	`\W`	cualquier carácter que no sea de palabra
`\s`	cualquier espacio en blanco	`\S`	cualquier carácter que no sea de espacio en blanco

Figura 30.19 | Clases predefinidas de caracteres.

```
// Fig. 30.20: ValidacionEntrada.java
// Valida la información del usuario mediante expresiones regulares.

public class ValidacionEntrada
{
   // valida el primer nombre
   public static boolean validarPrimerNombre( String primerNombre )
   {
      return primerNombre.matches( "[A-Z][a-zA-Z]*" );
   } // fin del método validarPrimerNombre

   // valida el apellido
   public static boolean validarApellidoPaterno( String apellidoPaterno )
   {
      return apellidoPaterno.matches( "[a-zA-z]+([ '-][a-zA-Z]+)*" );
   } // fin del método validarApellidoPaterno

   // valida la dirección
   public static boolean validarDireccion( String direccion )
   {
      return direccion.matches(
         "\\d+\\s+([a-zA-Z]+|[a-zA-Z]+\\s[a-zA-Z]+)" );
   } // fin del método validarDireccion

   // valida la ciudad
   public static boolean validarCiudad( String ciudad )
   {
      return ciudad.matches( "([a-zA-Z]+|[a-zA-Z]+\\s[a-zA-Z]+)" );
   } // fin del método validarCiudad
```

Figura 30.20 | Valida la información del usuario mediante expresiones regulares. (Parte 1 de 2).

```

   // valida el estado
   public static boolean validarEstado( String estado )
   {
      return estado.matches( "([a-zA-Z]+|[a-zA-Z]+\\s[a-zA-Z]+)" ) ;
   } // fin del método validarEstado

   // valida el código postal
   public static boolean validarCP( String cp )
   {
      return cp.matches( "\\d{5}" );
   } // fin del método validarCP

   // valida el teléfono
   public static boolean validarTelefono( String telefono )
   {
      return telefono.matches( "[1-9]\\d{2}-[1-9]\\d{2}-\\d{4}" );
   } // fin del método validarTelefono
} // fin de la clase ValidacionEntrada
```

Figura 30.20 | Valida la información del usuario mediante expresiones regulares. (Parte 2 de 2).

```
// Fig. 30.21: Validacion.java
// Valida la información del usuario mediante expresiones regulares.
import java.util.Scanner;

public class Validacion
{
   public static void main( String[] args )
   {
      // obtiene la entrada del usuario
      Scanner scanner = new Scanner( System.in );
      System.out.println( "Escriba el primer nombre:" );
      String primerNombre = scanner.nextLine();
      System.out.println( "Escriba el apellido paterno:" );
      String apellidoPaterno = scanner.nextLine();
      System.out.println( "Escriba la direccion:" );
      String direccion = scanner.nextLine();
      System.out.println( "Escriba la ciudad:" );
      String ciudad = scanner.nextLine();
      System.out.println( "Escriba el estado:" );
      String estado = scanner.nextLine();
      System.out.println( "Escriba el codigo postal:" );
      String cp = scanner.nextLine();
      System.out.println( "Escriba el telefono:" );
      String telefono = scanner.nextLine();

      // valida la entrada del usuario y muestra mensaje de error
      System.out.println( "\nValidar resultado:" );

      if ( !ValidacionEntrada.validarPrimerNombre( primerNombre ) )
         System.out.println( "Primer nombre invalido" );
      else if ( !ValidacionEntrada.validarApellidoPaterno( apellidoPaterno ) )
         System.out.println( "Apellido paterno invalido" );
      else if ( !ValidacionEntrada.validarDireccion( direccion ) )
         System.out.println( "Direccion invalida" );
      else if ( !ValidacionEntrada.validarCiudad( ciudad ) )
         System.out.println( "Ciudad invalida" );
```

Figura 30.21 | Recibe datos del usuario y los valida mediante la clase `ValidacionEntrada`. (Parte 1 de 2).

```
         else if ( !ValidacionEntrada.validarEstado( estado ) )
            System.out.println( "Estado invalido" );
         else if ( !ValidacionEntrada.validarCP( cp ) )
            System.out.println( "Codigo postal invalido" );
         else if ( !ValidacionEntrada.validarTelefono( telefono ) )
            System.out.println( "Numero telefonico invalido" );
         else
            System.out.println( "La entrada es valida.   Gracias." );
      } // fin de main
   } // fin de la clase Validacion
```

```
Escriba el primer nombre:
Jane
Escriba el apellido paterno:
Doe
Escriba la direccion:
123 Cierta Calle
Escriba la ciudad:
Una ciudad
Escriba el estado:
SS
Escriba el codigo postal:
123
Escriba el telefono:
123-456-7890

Validar resultado:
Codigo postal invalido
```

```
Escriba el primer nombre:
Jane
Escriba el apellido paterno:
Doe
Escriba la direccion:
123 Una calle
Escriba la ciudad:
Una ciudad
Escriba el estado:
SS
Escriba el codigo postal:
12345
Escriba el telefono:
123-456-7890

Validar resultado:
La entrada es valida.  Gracias.
```

Figura 30.21 | Recibe datos del usuario y los valida mediante la clase `ValidacionEntrada`. (Parte 2 de 2).

En la figura 30.20 se valida la entrada del usuario. En la línea 9 se valida el nombre. Para hacer que concuerde un conjunto de caracteres que no tiene una clase predefinida de carácter, utilice los corchetes (`[]`). Por ejemplo, el patrón `"[aeiou]"` puede utilizarse para concordar con una sola vocal. Los rangos de caracteres pueden representarse colocando un guión corto (`-`) entre dos caracteres. En el ejemplo, `"[A-Z]"` concuerda con una sola letra mayúscula. Si el primer carácter entre corchetes es `"^"`, la expresión acepta cualquier carácter distinto a los que se indiquen. Sin embargo, es importante observar que `"[^Z]"` no es lo mismo que `"[A-Y]"`, la cual concuerda con las letras mayúsculas `A-Y`; `"[^Z]"` concuerda con cualquier carácter distinto de la letra Z mayúscula, incluyendo las letras minúsculas y los caracteres que no son letras, como el carácter de nueva línea. Los rangos en

las clases de caracteres se determinan mediante los valores enteros de las letras. En este ejemplo, "[A-Za-z]" concuerda con todas las letras mayúsculas y minúsculas. El rango "[A-z]" concuerda con todas las letras y también concuerda con los caracteres (como % y 6) que tengan un valor entero entre la letra Z mayúscula y la letra a minúscula (para obtener más información acerca de los valores enteros de los caracteres, consulte el apéndice B, Conjunto de caracteres ASCII). Al igual que las clases predefinidas de caracteres, las clases de caracteres delimitadas entre corchetes concuerdan con un solo carácter en el objeto de búsqueda.

En la línea 9 (figura 30.20), el asterisco después de la segunda clase de carácter indica que puede concordar cualquier número de letras. En general, cuando aparece el operador de expresión regular "*" en una expresión regular, el programa intenta hacer que concuerden cero o más ocurrencias de la subexpresión que va inmediatamente después de "*". El operador "+" intenta hacer que concuerden una o más ocurrencias de la subexpresión que va inmediatamente después de "+". Por lo tanto, "A*" y "A+" concordarán con "AAA", pero sólo "A*" concordará con una cadena vacía.

Si el método validarPrimerNombre devuelve true (línea 29 de la figura 30.21), la aplicación trata de validar el apellido (línea 31) llamando a validarApellidoPaterno (líneas 13 a 16 de la figura 30.20). La expresión regular para validar el apellido concuerda con cualquier número de letras divididas por espacios, apóstrofes o guiones cortos.

En la línea 33 se valida la dirección, llamando al método validarDireccion (líneas 19 a 23 de la figura 30.20). La primera clase de carácter concuerda con cualquier dígito una o más veces (\\d+). Observe que se utilizan dos caracteres \, ya que \ generalmente inicia una secuencia de escape en una cadena. Por lo tanto, \\d en una cadena de Java representa al patrón de expresión regular \d. Después concordamos uno o más caracteres de espacio en blanco (\\s+). El carácter "|" concuerda con la expresión a su izquierda o a su derecha. Por ejemplo, "Hola (Juan | Juana)" concuerda tanto con "Hola Juan" como con "Hola Juana". Los paréntesis se utilizan para agrupar partes de la expresión regular. En este ejemplo, el lado izquierdo de | concuerda con una sola palabra y el lado derecho concuerda con dos palabras separadas por cualquier cantidad de espacios en blanco. Por lo tanto, la dirección debe contener un número seguido de una o dos palabras. Por lo tanto, "10 Broadway" y "10 Main Street" son ambas direcciones válidas en este ejemplo. Los métodos ciudad (líneas 26 a 29 de la figura 30.20) y estado (líneas 32 a 35 de la figura 30.20) también concuerdan con cualquier palabra que tenga al menos un carácter o, de manera alternativa, con dos palabras cualesquiera con al menos un carácter, si éstas van separadas por un solo espacio. Esto significa que tanto Waltham como West Newton concordarían.

Cuantificadores

El asterisco (*) y el signo de suma (+) se conocen de manera formal como **cuantificadores**. En la figura 30.22 se presentan todos los cuantificadores. Ya hemos visto cómo funcionan el asterisco (*) y el signo de suma (+). Todos los cuantificadores afectan solamente a la subexpresión que va inmediatamente antes del cuantificador. El cuantificador signo de interrogación (?) concuerda con cero o una ocurrencia de la expresión que cuantifica. Un conjunto de llaves que contienen un número ({*n*}) concuerda exactamente con *n* ocurrencias de la expresión que cuantifica. En la figura 30.20 mostramos este cuantificador para validar el código postal, en la línea 40. Si se incluye una coma después del número encerrado entre llaves, el cuantificador concordará al menos con n ocurrencias de la expresión cuantificada. El conjunto de llaves que contienen dos números ({*n*,*m*}) concuerda entre *n* y *m* ocurrencias de la expresión que califica. Los cuantificadores pueden aplicarse a patrones encerrados entre paréntesis para crear expresiones regulares más complejas.

Todos los cuantificadores son **avaros**. Esto significa que concordarán con todas las ocurrencias que puedan, siempre y cuando haya concordancia. No obstante, si alguno de estos cuantificadores va seguido por un signo de interrogación (?), el cuantificador se vuelve **reacio** (o, en algunas ocasiones, **flojo**). De esta forma, concordará con la menor cantidad de ocurrencias posibles, siempre y cuando haya concordancia.

El código postal (línea 40 en la figura 30.20) concuerda con un dígito cinco veces. Esta expresión regular utiliza la clase de carácter de dígito y un cuantificador con el dígito 5 entre llaves. El número telefónico (línea 46 en la figura 30.20) concuerda con tres dígitos (el primero no puede ser cero) seguidos de un guión corto, seguido de tres dígitos más (de nuevo, el primero no puede ser cero), seguidos de cuatro dígitos más.

El método matches de String verifica si una cadena completa se conforma a una expresión regular. Por ejemplo, queremos aceptar "Smith" como apellido, pero no "9@Smith#". Si sólo una subcadena concuerda con la expresión regular, el método matches devuelve false.

Cuantificador	Concuerda con
`*`	Concuerda con cero o más ocurrencias del patrón.
`+`	Concuerda con una o más ocurrencias del patrón.
`?`	Concuerda con cero o una ocurrencia del patrón.
`{n}`	Concuerda con exactamente `n` ocurrencias.
`{n,}`	Concuerda con al menos `n` ocurrencias.
`{n,m}`	Concuerda con entre `n` y `m` (inclusive) ocurrencias.

Figura 30.22 | Cuantificadores utilizados en expresiones regulares.

Reemplazo de subcadenas y división de cadenas

En ocasiones es conveniente reemplazar partes de una cadena, o dividir una cadena en varias piezas. Para este fin, la clase `String` proporciona los métodos **`replaceAll`**, **`replaceFirst`** y **`split`**. Estos métodos se muestran en la figura 30.23.

```
// Fig. 30.23: SustitucionRegex.java
// Uso de los métodos replaceFirst, replaceAll y split.

public class SustitucionRegex
{
   public static void main( String args[] )
   {
      String primeraCadena = "Este enunciado termina con 5 estrellas *****";
      String segundaCadena = "1, 2, 3, 4, 5, 6, 7, 8";

      System.out.printf( "Cadena 1 original: %s\n", primeraCadena );

      // sustituye '*' con '^'
      primeraCadena = primeraCadena.replaceAll( "\\*", "^" );

      System.out.printf( "^ sustituyen a *: %s\n", primeraCadena );

      // sustituye 'estrellas' con 'ntercaladores'
      primeraCadena = primeraCadena.replaceAll( "estrellas", "intercaladores" );

      System.out.printf(
         "\"intercaladores\" sustituye a \"estrellas\": %s\n", primeraCadena );

      // sustituye las palabras con 'palabra'
      System.out.printf( "Cada palabra se sustituye por \"palabra\": %s\n\n",
         primeraCadena.replaceAll( "\\w+", "palabra" ) );

      System.out.printf( "Cadena 2 original: %s\n", segundaCadena );

      // sustituye los primeros tres dígitos con 'digito'
      for ( int i = 0; i < 3; i++ )
         segundaCadena = segundaCadena.replaceFirst( "\\d", "digito" );

      System.out.printf(
         "Los primeros 3 digitos se sustituyeron por \"digito\" : %s\n", segundaCadena );
```

Figura 30.23 | Métodos `replaceFirst`, `replaceAll` y `Split`. (Parte 1 de 2).

```
36        String salida = "Cadena dividida en comas: [";
37
38        String[] resultados = segundaCadena.split( ",\\s*" ); // se divide en las comas
39
40        for ( String cadena : resultados )
41           salida += "\"" + cadena + "\", "; // imprime resultados
42
43        // elimina la coma adiconal y agrega un corchete
44        salida = salida.substring( 0, salida.length() - 2 ) + "]";
45        System.out.println( salida );
46     } // fin de main
47  } // fin de la clase SustitucionRegex
```

```
Cadena 1 original: Este enunciado termina con 5 estrellas *****
^ sustituyen a *: Este enunciado termina con 5 estrellas ^^^^^
"intercaladores" sustituye a "estrellas": Este enunciado termina con 5 intercaladores ^^^^^
Cada palabra se sustituye por "palabra": palabra palabra palabra palabra palabra palabra
^^^^^

Cadena 2 original: 1, 2, 3, 4, 5, 6, 7, 8
Los primeros 3 digitos se sustituyeron por "digito" : digito, digito, digito, 4, 5, 6, 7, 8
Cadena dividida en comas: ["digito", "digito", "digito", "4", "5", "6", "7", "8"]
```

Figura 30.23 | Métodos `replaceFirst`, `replaceAll` y `Split`. (Parte 2 de 2).

El método `replaceAll` reemplaza el texto en una cadena con nuevo texto (el segundo argumento) en cualquier parte en donde la cadena original concuerde con una expresión regular (el primer argumento). En la línea 14 se reemplaza cada instancia de `"*"` en `primeraCadena` con `"^"`. Observe que la expresión regular (`"\\*"`) coloca dos barras diagonales inversas (\) antes del carácter *. Por lo general, * es un cuantificador que indica que una expresión regular debe concordar con cualquier número de ocurrencias del patrón que se coloca antes de este carácter. Sin embargo, en la línea 14 queremos encontrar todas las ocurrencias del carácter literal *; para ello, debemos evadir el carácter * con el carácter \. Al evadir un carácter especial de expresión regular con una ****, indicamos al motor de concordancia de expresiones regulares que busque el carácter en sí. Como la expresión está almacenada en una cadena de Java y \ es un carácter especial en las cadenas de Java, debemos incluir un **** adicional. Por lo tanto, la cadena de Java `"\\*"` representa el patrón de expresión regular `\*`, que concuerda con un solo carácter * en la cadena de búsqueda. En la línea 19, todas las coincidencias con la expresión regular `"estrellas"` en `primeraCadena` se reemplazan con `"intercaladores"`.

El método `replaceFirst` (línea 32) reemplaza la primera ocurrencia de la concordancia de un patrón. Las cadenas de Java son inmutables, por lo cual el método `replaceFirst` devuelve una nueva cadena en la que se han reemplazado los caracteres apropiados. Esta línea toma la cadena original y la reemplaza con la cadena devuelta por `replaceFirst`. Al iterar tres veces, reemplazamos las primeras tres instancias de un dígito (`\d`) en `segundaCadena` con el texto `"digito"`.

El método `split` divide una cadena en varias subcadenas. La cadena original se divide en cualquier posición que concuerde con una expresión regular especificada. El método `split` devuelve un arreglo de cadenas que contiene las subcadenas que resultan de cada concordancia con la expresión regular. En la línea 38 utilizamos el método `split` para descomponer en tokens una cadena de enteros separados por comas. El argumento es la expresión regular que localiza el delimitador. En este caso, utilizamos la expresión regular `",\\s*"` para separar las subcadenas siempre que haya una coma. Al concordar con cualquier carácter de espacio en blanco, eliminamos los espacios adicionales de las subcadenas resultantes. Observe que las comas y los espacios en blanco no se devuelven como parte de las subcadenas. De nuevo, observe que la cadena de Java `",\\s*"` representa la expresión regular `,\s*`.

Las clases `Pattern` *y* `Matcher`

Además de las herramientas para el uso de expresiones regulares de la clase `String`, Java proporciona otras clases en el paquete `java.util.regex` que ayudan a los desarrolladores a manipular expresiones regulares. La clase

Pattern representa una expresión regular. La clase **Matcher** contiene tanto un patrón de expresión regular como un objeto CharSequence en el que se va a buscar ese patrón.

CharSequence es una interfaz que permite el acceso de lectura a una secuencia de caracteres. Esta interfaz requiere que se declaren los métodos charAt, length, subSequence y toString. Tanto String como StringBuilder implementan la interfaz CharSequence, por lo que puede usarse una instancia de cualquiera de estas clases con la clase Matcher.



Error común de programación 30.4

Una expresión regular puede compararse con un objeto de cualquier clase que implemente a la interfaz CharSequence, pero la expresión regular debe ser un objeto String. Si se intenta crear una expresión regular como un objeto StringBuilder se produce un error.

Si se va a utilizar una expresión regular sólo una vez, puede usarse el método static **matches** de la clase Pattern. Este método toma una cadena que especifica la expresión regular y un objeto CharSequence en la que se va a realizar la prueba de concordancia. Este método devuelve un valor de tipo boolean, el cual indica si el objeto de búsqueda (el segundo argumento) concuerda con la expresión regular.

Si se va a utilizar una expresión regular más de una vez, es más eficiente usar el método static **compile** de la clase Pattern para crear un objeto Pattern específico de esa expresión regular. Este método recibe una cadena que representa el patrón y devuelve un nuevo objeto Pattern, el cual puede utilizarse para llamar al método **matcher**. Este método recibe un objeto CharSequence en el que se va a realizar la búsqueda, y devuelve un objeto Matcher.

La clase Matcher cuenta con el método **matches**, el cual realiza la misma tarea que el método matches de Pattern, pero no recibe argumentos; el patrón y el objeto de búsqueda están encapsulados en el objeto Matcher. La clase Matcher proporciona otros métodos, incluyendo **find**, **lookingAt**, **replaceFirst** y **replaceAll**.

En la figura 30.24 presentamos un ejemplo sencillo en el que se utilizan expresiones regulares. Este programa compara las fechas de cumpleaños con una expresión regular. La expresión concuerda sólo con los cumpleaños que no ocurran en abril y que pertenezcan a personas cuyos nombres empiecen con "J".

```
// Fig. 30.24: ConcordanciasRegex.java
// Demostración de las clases Pattern y Matcher.
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class ConcordanciasRegex
{
   public static void main( String args[] )
   {
      // crea la expresión regular
      Pattern expresion =
         Pattern.compile( "J.*\\d[0-35-9]-\\d\\d-\\d\\d" );

      String cadena1 = "Jane nacio el 05-12-75\n" +
         "Dave nacio el 11-04-68\n" +
         "John nacio el 04-28-73\n" +
         "Joe nacio el 12-17-77";

      // compara la expresión regular con la cadena e imprime las concordancias
      Matcher matcher = expresion.matcher( cadena1 );

      while ( matcher.find() )
         System.out.println( matcher.group() );
   } // fin de main
} // fin de la clase ConcordanciasRegex
```

Figura 30.24 | Expresiones regulares para verificar fechas de nacimiento. (Parte 1 de 2).

```
Jane nacio el 05-12-75
Joe nacio el 12-17-77
```

Figura 30.24 | Expresiones regulares para verificar fechas de nacimiento. (Parte 2 de 2).

En las líneas 11 y 12 se crea un objeto `Pattern` mediante la invocación al método estático `compile` de la clase `Pattern`. El carácter de punto `"."` en la expresión regular (línea 12) concuerda con cualquier carácter individual, excepto un carácter de nueva línea.

En la línea 20 se crea el objeto `Matcher` para la expresión regular compilada y la secuencia de concordancia (`cadena1`). En las líneas 22 y 23 se utiliza un ciclo `while` para iterar a través de la cadena. En la línea 22 se utiliza el método `find` de la clase `Matcher` para tratar de hacer que concuerde una pieza del objeto de búsqueda con el patrón de búsqueda. Cada una de las llamadas a este método empieza en el punto en el que terminó la última llamada, por lo que pueden encontrarse varias concordancias. El método `lookingAt` de la clase `Matcher` funciona de manera similar, sólo que siempre comienza desde el principio del objeto de búsqueda, y siempre encontrará la primera concordancia, si es que hay una.



Error común de programación 30.5

El método `matches` (de las clases `String`, `Pattern` o `Matcher`) devuelve `true` sólo si todo el objeto de búsqueda concuerda con la expresión regular. Los métodos `find` y `lookingAt` (de la clase `Matcher`) devuelven `true` si una parte del objeto de búsqueda concuerda con la expresión regular.

En la línea 23 se utiliza el método **`group`** de la clase `Matcher`, el cual devuelve la cadena del objeto de búsqueda que concuerda con el patrón de búsqueda. La cadena devuelta es la que haya concordado la última vez en una llamada a `find` o `lookingAt`. La salida en la figura 30.24 muestra las dos concordancias que se encontraron en `cadena1`.

Recursos Web sobre expresiones regulares

Los siguientes sitios Web proporcionan más información sobre las expresiones regulares.

`java.sun.com/docs/books/tutorial/extra/regex/index.html`
Este tutorial explica cómo utilizar el API de expresiones regulares de Java.

`java.sun.com/javase/6/docs/api/java/util/regex/package-summary.html`
Esta página es el panorama general del paquete `java.util.regex` mediante el uso de javadoc.

`developer.java.sun.com/developer/technicalArticles/releases/1.4regex`
Este sitio incluye una descripción detallada de las herramientas para expresiones regulares del lenguaje Java.

30.8 Conclusión

En este capítulo aprendió acerca de más métodos de `String` para seleccionar porciones de objetos `String` y manipularlos. También aprendió acerca de la clase `Character` y sobre algunos de los métodos que declara para manejar valores `char`. En este capítulo también hablamos sobre las herramientas de la clase `StringBuilder` para crear objetos `String`. En la parte final del capítulo hablamos sobre las expresiones regulares, las cuales proporcionan una poderosa herramienta para buscar y relacionar porciones de objetos `String` que coincidan con un patrón específico.

Resumen

Sección 30.2 Fundamentos de los caracteres y las cadenas

- El valor de una literal de carácter es el valor entero del carácter en el conjunto de caracteres Unicode. Las cadenas pueden incluir letras, dígitos y varios caracteres especiales, tales como `+`, `-`, `*`, `/` y `$`. Una cadena en Java es un objeto de la clase `String`. Las literales de cadena se conocen, por lo regular, como objetos `String`, y se escriben entre comillas dobles en un programa.

Sección 30.3 La clase `String`

- Los objetos `String` son inmutables: los caracteres que contienen no se pueden modificar una vez que se crean.
- El método `length` de `String` devuelve el número de caracteres en un objeto `String`.
- El método `charAt` de `String` devuelve el carácter en una posición específica.
- El método `equals` de `String` compara la igualdad entre dos objetos. Este método devuelve `true` si el contenido de los objetos `String` es igual, y `false` en caso contrario. El método `equals` utiliza una comparación lexicográfica para los objetos `String`.
- Cuando se comparan valores de tipo primitivo con `==`, el resultado es `true` si ambos valores son idénticos. Cuando las referencias se comparan con `==`, el resultado es `true` si ambas referencias son al mismo objeto en memoria.
- Java trata a todas las literales de cadena con el mismo contenido como un solo objeto `String`.
- El método `equalsIgnoreCase` de `String` realiza una comparación de cadenas insensible al uso de mayúsculas y minúsculas.
- El método `compareTo` de `String` usa una comparación lexicográfica y devuelve 0 si las cadenas que está comparando son iguales, un número negativo si la cadena con la que se invoca a `compareTo` es menor que el objeto `String` que recibe como argumento, y un número positivo si la cadena con la que se invoca a `compareTo` es mayor que la cadena que recibe como argumento.
- El método `regionMatches` de `String` compara la igualdad entre porciones de dos cadenas.
- El método `startsWith` de `String` determina si una cadena empieza con los caracteres especificados como argumento. El método `endsWith` de `String` determina si una cadena termina con los caracteres especificados como argumento.
- El método `indexOf` de `String` localiza la primera ocurrencia de un carácter, o de una subcadena en una cadena. El método `lastIndexOf` de `String` localiza la última ocurrencia de un carácter, o de una subcadena en una cadena.
- El método `substring` de `String` copia y devuelve parte de un objeto cadena existente.
- El método `concat` de `String` concatena dos objetos cadena y devuelve un nuevo objeto cadena, que contiene los caracteres de ambas cadenas originales.
- El método `replace` de `String` devuelve un nuevo objeto cadena que reemplaza cada ocurrencia en un objeto `String` de su primer argumento carácter, con su segundo argumento carácter.
- El método `toUpperCase` de `String` devuelve una nueva cadena con letras mayúsculas, en las posiciones en donde la cadena original tenía letras minúsculas. El método `toLowerCase` de `String` devuelve una nueva cadena con letras minúsculas en las posiciones en donde la cadena original tenía letras mayúsculas.
- El método `trim` de `String` devuelve un nuevo objeto cadena, en el que todos los caracteres de espacio en blanco (espacios, nuevas líneas y tabuladores) se eliminan de la parte inicial y la parte final de una cadena.
- El método `toCharArray` de `String` devuelve un arreglo `char` que contiene una copia de los caracteres de una cadena.
- El método `static valueOf` de `String` devuelve su argumento convertido en una cadena.

Sección 30.4 La clase `StringBuilder`

- La clase `StringBuilder` proporciona constructores que permiten inicializar objetos `StringBuilders` sin caracteres, y con una capacidad inicial de 16 caracteres, sin caracteres y con una capacidad inicial especificada en el argumento entero, o con una copia de los caracteres del argumento `String` y una capacidad inicial equivalente al número de caracteres en el argumento `String`, más 16.
- El método `length` de `StringBuilder` devuelve el número de caracteres actualmente almacenados en un objeto `StringBuilder`. El método `capacity` de `StringBuilder` devuelve el número de caracteres que se pueden almacenar en un objeto `StringBuilder` sin necesidad de asignar más memoria.
- El método `ensureCapacity` de `StringBuilder` asegura que un objeto `StringBuilder` tenga por lo menos la capacidad especificada. El método `setLength` de `StringBuilder` incrementa o decrementa la longitud de un objeto `StringBuilder`.
- El método `charAt` de `StringBuilder` devuelve el carácter que se encuentra en el índice especificado. El método `setCharAt` de `StringBuilder` establece el carácter en la posición especificada. El método `getChars` de `StringBuilder` copia los caracteres que están en el objeto `StringBuilder` y los coloca en el arreglo de caracteres que se pasa como argumento.
- La clase `StringBuilder` proporciona métodos `append` para agregar valores de tipo primitivo, arreglos de caracteres, `String`, `Object` y `CharSequence` al final de un objeto `StringBuilder`. El compilador de Java utiliza los objetos `StringBuilder` y los métodos `append` para implementar los operadores de concatenación `+` y `+=`.
- La clase `StringBuilder` proporciona métodos `insert` sobrecargados para insertar valores de tipo primitivo, arreglos de caracteres, `String`, `Object` y `CharSequence` en cualquier posición en un objeto `StringBuilder`.

Sección 30.5 La clase **Character**

- La clase Character proporciona un constructor que recibe un argumento char.
- El método isDefined de Character determina si un carácter está definido en el conjunto de caracteres Unicode. De ser así, el método devuelve true; en caso contrario, devuelve false.
- El método isDigit de Character determina si un carácter es un dígito definido en Unicode. De ser así, el método devuelve true; en caso contrario devuelve false.
- El método isJavaIdentifierStart de Character determina si un carácter se puede utilizar como el primer carácter de un identificador en Java [es decir, una letra, un guión bajo (_) o un signo de dólar ($)]. De ser así, el método devuelve true; en caso contrario, devuelve false.
- El método isJavaIdentifierPart de Character determina si se puede utilizar un carácter en un identificador en Java [es decir, un dígito, una letra, un guión bajo (_) o un signo de dólar ($)]. El método isLetter de Character determina si un carácter es una letra. El método isLetterOrDigit de Character determina si un carácter es una letra o un dígito. En cada caso, si es así, el método devuelve true; en caso contrario devuelve false.
- El método isLowerCase de Character determina si un carácter es una letra minúscula. El método isUpperCase de Character determina si un carácter es una letra mayúscula. En ambos casos, de ser así el método devuelve true; en caso contrario devuelve false.
- El método toUpperCase de Character convierte un carácter en su equivalente en mayúscula. El método toLowerCase convierte un carácter en su equivalente en minúscula.
- El método digit de Character convierte su argumento carácter en un entero en el sistema numérico especificado por su argumento entero raiz. El método forDigit de Character convierte su argumento entero digito en un carácter en el sistema numérico especificado por su argumento entero raiz.
- El método charValue de Character devuelve el valor char almacenado en un objeto Character. El método toString de Characer devuelve una representación String de un objeto Character.

Sección 30.6 La clase **StringTokenizer**

- El constructor predeterminado de StringTokenizer crea un objeto StringTokenizer para su argumento cadena que utilizará la cadena delimitadora predeterminada "\t\n\r\f", la cual consiste en un espacio, un tabulador, un carácter de nueva línea y un retorno de carro, para dividir la cadena en tokens.
- El método countTokens de StringTokenizer devuelve el número de tokens en una cadena que se va a dividir en tokens.
- El método hasMoreTokens de StringTokenizer determina si hay más tokens en la cadena que se va a dividir en tokens.
- El método nextToken de StringTokenizer devuelve un objeto String con el siguiente token.

Sección 30.7 Expresiones regulares, la clase **Pattern** *y la clase* **Matcher**

- Las expresiones regulares son secuencias de caracteres y símbolos que definen un conjunto de cadenas. Son útiles para validar la entrada y asegurar que lo datos se encuentren en un formato específico.
- El método matches de String recibe una cadena que especifica una expresión regular y relaciona el contenido del objeto String en el que se llama con la expresión regular. El método devuelve un valor de tipo boolean, el cual indica si hubo concordancia o no.
- Una clase de carácter es una secuencia de escape que representa a un grupo de caracteres. Cada clase de carácter concuerda con un solo carácter en la cadena que estamos tratando de igualar con la expresión regular.
- Un carácter de palabra (\w) es cualquier letra (mayúscula o minúscula), dígito o el carácter de guión bajo.
- Un carácter de espacio en blanco (\s) es un espacio, un tabulador, un retorno de carro, un carácter de nueva línea o un avance de página.
- Un dígito (\d) es cualquier carácter numérico.
- Para relacionar un conjunto de caracteres que no tienen una clase de carácter predefinida, use corchetes ([]). Para representar los rangos, coloque un guión corto (-) entre dos caracteres. Si el primer carácter en los corchetes es "^", la expresión acepta a cualquier carácter distinto de los que se indican.
- Cuando aparece el operador "*" en una expresión regular, el programa trata de relacionar cero o más ocurrencias de la subexpresión que está justo antes del "*".
- El operador "+" trata de relacionar una o más ocurrencias de la subexpresión que está antes de éste.
- El carácter "|" permite una concordancia de la expresión a su izquierda o a su derecha.
- Los paréntesis ( ) se utilizan para agrupar partes de la expresión regular.
- El asterisco (*) y el signo positivo (+) se conocen formalmente como cuantificadores.
- Todos los cuantificadores afectan sólo a la subexpresión que va justo antes del cuantificador.
- El cuantificador signo de interrogación (?) concuerda con cero o una ocurrencias de la expresión que cuantifica.

- Un conjunto de llaves que contienen un número ({n}) concuerda exactamente con n ocurrencias de la expresión que cuantifica. Si se incluye una coma después del número encerrado entre llaves, concuerda con al menos n ocurrencias de la expresión cuantificada.
- Un conjunto de llaves que contienen dos números ({n,m}) concuerda con entre n y m ocurrencias de la expresión que califica.
- Todos los cuantificadores son avaros, lo cual significa que concordarán con tantas ocurrencias como puedan, mientras que haya concordancia.
- Si cualquiera de estos cuantificadores va seguido de un signo de interrogación (?), el cuantificador se vuelve renuente, y concuerda con el menor número posible de ocurrencias, mientras que haya concordancia.
- El método replaceAll de String reemplaza texto en una cadena con nuevo texto (el segundo argumento), en cualquier parte en donde la cadena original concuerde con una expresión regular (el primer argumento).
- Al escapar un carácter de expresión regular especial con una \, indicamos al motor de concordancia de expresiones regulares que encuentre el carácter actual, en contraste a lo que representa en una expresión regular.
- El método replaceFirst de String reemplaza la primera ocurrencia de la concordancia de un patrón. Los objetos String de Java son inmutables, por lo cual el método replaceFirst devuelve una nueva cadena en la que se han reemplazado los caracteres apropiados.
- El método split de String divide una cadena en varias subcadenas. La cadena original se divide en cualquier ubicación que concuerde con una expresión regular especificada. El método split devuelve un arreglo de cadenas que contienen las subcadenas entre las concordancias para la expresión regular.
- La clase Pattern representa a una expresión regular.
- La clase Matcher contiene tanto un patrón de expresión regular como un objeto CharSequence, en el cual puede buscar el patrón.
- CharSequence es una interfaz que permite el acceso de lectura a una secuencia de caracteres. Tanto String como StringBuilder implementan a la interfaz CharSequence, por lo que se puede utilizar una instancia de cualquiera de estas clases con la clase Matcher.
- Si una expresión regular se va a utilizar sólo una vez, el método estático matches de Pattern recibe una cadena que especifica la expresión regular y un objeto CharSequence en el que se va a realizar la concordancia. Este método devuelve un valor de tipo boolean que indica si el objeto de búsqueda concuerda o no con la expresión regular.
- Si una expresión regular se va a utilizar más de una vez, es más eficiente usar el método estático compile de Pattern para crear un objeto Pattern especifico para esa expresión regular. Este método recibe una cadena que representa el patrón y devuelve un nuevo objeto Pattern.
- El método matcher de Pattern recibe un objeto CharSequence para realizar la búsqueda y devuelve un objeto Matcher.
- El método matches de Matcher realiza la misma tarea que el método matches de Pattern, pero no recibe argumentos.
- El método find de Matcher trata de relacionar una pieza del objeto de la búsqueda con el patrón de búsqueda. Cada llamada a este método empieza en el punto en el que terminó la última llamada, por lo que se pueden encontrar varias concordancias.
- El método lookingAt de Matcher realiza lo mismo que find, excepto que siempre empieza desde el inicio del objeto de búsqueda, y siempre encuentra la primera concordancia, si hay una.
- El método group de Matcher devuelve la cadena del objeto de búsqueda que concuerda con el patrón de búsqueda. La cadena que se devuelve es la última que concordó mediante una llamada a find o a lookingAt.

Terminología

append, método de la clase StringBuilder
cadena vacía
capacity, método de la clase StringBuilder
carácter de palabra
carácter especial
charAt, método de la clase StringBuilder
CharSequence, interfaz
charValue, método de la clase Character
clase de carácter predefinida
comparación lexicográfica
concat, método de la clase String
cuantificador avaro
cuantificador flojo
cuantificador para expresión regular
cuantificador renuente
delete, método de la clase StringBuilder
deleteCharAt, método de la clase String
delimitador para tokens
digit, método de la clase Character
endsWith, método de la clase String
ensureCapacity, método de la clase StringBuilder
expresiones regulares

find, método de la clase Matcher
forDigit, método de la clase Character
getChars, método de la clase String
getChars, método de la clase StringBuilder
hasMoreTokens, método de la clase StringTokenizer
indexOf, método de la clase String
inmutable
isDefined, método de la clase Character
isDigit, método de la clase Character
isJavaIdentifierPart, método de la clase Character
isJavaIdentifierStart, método de la clase Character
isLetter, método de la clase Character
isLetterOrDigit, método de la clase Character
isLowerCase, método de la clase Character
isUpperCase, método de la clase Character
lastIndexOf, método de la clase String
length, método de la clase String
length, método de la clase StringBuilder
literal de cadena
literal de carácter
lookingAt, método de la clase Matcher
Matcher, clase
matcher, método de la clase Pattern
matches, método de la clase Matcher
matches, método de la clase Pattern
matches, método de la clase String
nextToken, método de la clase StringTokenizer
Pattern, clase
raíz
regionMatches, método de la clase String
replaceAll, método de la clase String
replaceFirst, método de la clase String
reverse, método de la clase StringBuilder
setCharAt, método de la clase StringBuilder
split, método de la clase String
startsWith, método de la clase String
StringIndexOutOfBoundsException, clase
token de un objeto String
toLowerCase, método de la clase Character
toUpperCase, método de la clase Character
trim, método de la clase StringBuilder
Unicode, conjunto de caracteres
valueOf, método de la clase String

Ejercicios de autoevaluación

30.1 Conteste con *verdadero* o *falso* a cada una de las siguientes proposiciones; en caso de ser *falso*, explique por qué.

a) Cuando los objetos String se comparan utilizando ==, el resultado es true si los objetos String contiene los mismos valores.
b) Un objeto String puede modificarse una vez creado.

30.2 Para cada uno de los siguientes enunciados, escriba una instrucción que realice la tarea indicada:

a) Comparar la cadena en s1 con la cadena en s2 para ver si su contenido es igual.
b) Anexar la cadena s2 a la cadena s1, utilizando +=.
c) Determinar la longitud de la cadena en s1.

Respuestas a los ejercicios de autoevaluación

30.1 a) Falso. Los objetos String se comparan con el operador == para determinar si son el mismo objeto en la memoria.
b) Falso. Los objetos String son inmutables y no pueden modificarse una vez creados. Los objetos StringBuilder sí pueden modificarse una vez creados.

30.2 a) `s1.equals( s1 )`
b) `s1 += s2;`
c) `s1.length()`

Ejercicios

30.3 Escriba una aplicación que utilice el método compareTo de la clase String para comparar dos cadenas introducidas por el usuario. Muestre si la primera cadena es menor, igual o mayor que la segunda.

30.4 Escriba una aplicación que utilice el método regionMatches de la clase String para comparar dos cadenas introducidas por el usuario. La aplicación deberá recibir como entrada el número de caracteres a comparar y el índice inicial de la comparación. La aplicación deberá indicar si las cadenas son iguales. Ignore si los caracteres están en mayúsculas o en minúsculas al momento de realizar la comparación.

30.5 Escriba una aplicación que utilice la generación de números aleatorios para crear enunciados. Use cuatro arreglos de cadenas llamados articulo, sustantivo, verbo y preposicion. Cree una oración seleccionando una palabra al azar de cada uno de los arreglos, en el siguiente orden: articulo, sustantivo, verbo, preposicion, articulo y

`sustantivo`. A medida que se elija cada palabra, concaténela con las palabras anteriores en el enunciado. Las palabras deberán separarse mediante espacios. Cuando se muestre el enunciado final, deberá empezar con una letra mayúscula y terminar con un punto. El programa deberá generar 20 enunciados y mostrarlos en un área de texto.

30.6 El arreglo de artículos debe contener los artículos `"el"`, `"un"`, `"algún"` y `"ningún"`; el arreglo de sustantivos deberá contener los sustantivos `"ninio"`, `"ninia"`, `"perro"`, `"ciudad"` y `"auto"`; el arreglo de verbos deberá contener los verbos `"manejo"`, `"salto"`, `"corrio"`, `"camino"` y `"omitio"`; el arreglo de preposiciones deberá contener las preposiciones `"a"`, `"desde"`, `"encima de"`, `"debajo de"` y `"sobre"`.

30.7 Una vez que escriba el programa anterior, modifíquelo para producir una historia breve que consista de varias de estas oraciones (¿qué hay sobre la posibilidad de un escritor de exámenes finales al azar?)

30.8 (*Quintillas)* Una quintilla es un verso humorístico de cinco líneas en el cual la primera y segunda línea riman con la quinta, y la tercera línea rima con la cuarta. Utilizando técnicas similares a las desarrolladas en el ejercicio 30.5, escriba una aplicación en Java que produzca quintillas al azar. Mejorar el programa para producir buenas quintillas es un gran desafío, ¡pero el resultado valdrá la pena!

30.9 *(Latín cerdo)* Escriba una aplicación que codifique frases en español a frases en latín cerdo. El latín cerdo es una forma de lenguaje codificado. Existen muchas variaciones en los métodos utilizados para formar frases en latín cerdo. Por cuestiones de simpleza, utilice el siguiente algoritmo:

Para formar una frase en latín cerdo a partir de una frase en español, divida la frase en palabras con un objeto de la clase `StringTokenizer`. Para traducir cada palabra en español a una palabra en latín cerdo, coloque la primera letra de la palabra en español al final de la palabra, y agregue las letras "ae". De esta forma, la palabra "salta" se convierte a "altasae", la palabra "el" se convierte en "leae" y la palabra "computadora" se convierte en "omputadoracae". Los espacios en blanco entre las palabras permanecen como espacios en blanco. Suponga que la frase en español consiste en palabras separadas por espacios en blanco, que no hay signos de puntuación y que todas las palabras tienen dos o más letras. El método `imprimirPalabraEnLatin` deberá mostrar cada palabra. Cada token devuelto de `nextToken` se pasará al método `imprimirPalabraEnLatin` para imprimir la palabra en latín cerdo. Permita al usuario introducir el enunciado. Use un área de texto para ir mostrando cada uno de los enunciados convertidos.

30.10 Escriba una aplicación que reciba como entrada un número telefónico como una cadena de la forma `(555) 555-5555`. La aplicación deberá utilizar un objeto de la clase `StringTokenizer` para extraer el código de área como un token, los primeros tres dígitos del número telefónico como otro token y los últimos cuatro dígitos del número telefónico como otro token. Los siete dígitos del número telefónico deberán concatenarse en una cadena. Deberán imprimirse tanto el código de área como el número telefónico. Recuerde que tendrá que modificar los caracteres delimitadores al dividir la cadena en tokens.

30.11 Escriba una aplicación que reciba como entrada una línea de texto, que divida la línea en tokens mediante un objeto de la clase `StringTokenizer` y que muestre los tokens en orden inverso. Use caracteres de espacio como delimitadores.

30.12 Use los métodos de comparación de cadenas que se describieron en este capítulo, junto con las técnicas para ordenar arreglos que se desarrollaron en el capítulo 16 para escribir una aplicación que ordene alfabéticamente una lista de cadenas. Permita al usuario introducir las cadenas en un campo de texto. Muestre los resultados en un área de texto.

30.13 Escriba una aplicación que reciba como entrada una línea de texto y que la imprima dos veces; una vez en letras mayúsculas y otra en letras minúsculas.

30.14 Escriba una aplicación que reciba como entrada una línea de texto y un carácter de búsqueda, y que utilice el método `indexOf` de la clase `String` para determinar el número de ocurrencias de ese carácter en el texto.

30.15 Escriba una aplicación con base en el programa del ejercicio 30.14, que reciba como entrada una línea de texto y utilice el método `indexOf` de la clase `String` para determinar el número total de ocurrencias de cada letra del alfabeto en ese texto. Las letras mayúsculas y minúsculas deben contarse como una sola. Almacene los totales para cada letra en un arreglo, e imprima los valores en formato tabular después de que se hayan determinado los totales.

30.16 Escriba una aplicación que lea una línea de texto, que divida la línea en tokens utilizando caracteres de espacio como delimitadores, y que imprima sólo aquellas palabras que comiencen con la letra `"b"`.

30.17 Escriba una aplicación que lea una línea de texto, que divida la línea en tokens utilizando caracteres de espacio como delimitadores, y que imprima sólo aquellas palabras que comiencen con las letras `"ED"`.

30.18 Escriba una aplicación que reciba como entrada un código entero para un carácter y que muestre el carácter correspondiente. Modifique esta aplicación de manera que genere todos los posibles códigos de tres dígitos en el rango de 000 a 255, y que intente imprimir los caracteres correspondientes.

30.19 Escriba sus propias versiones de los métodos de búsqueda `indexOf` y `lastIndexOf` de la clase `String`.

30.20 Escriba un programa que lea una palabra de cinco letras proveniente del usuario, y que produzca todas las posibles cadenas de tres letras que puedan derivarse de las letras de la palabra con cinco letras. Por ejemplo, las palabras de tres letras producidas a partir de la palabra "trigo" son "rio", "tio" y "oir".

Sección especial: manipulación avanzada de cadenas

Los siguientes ejercicios son clave para el libro y están diseñados para evaluar la comprensión del lector sobre los conceptos fundamentales de la manipulación de cadenas. Esta sección incluye una colección de ejercicios intermedios y avanzados de manipulación de cadenas. El lector encontrará estos ejercicios desafiantes, pero divertidos. Los problemas varían considerablemente en dificultad. Algunos requieren una hora o dos para escribir e implementar la aplicación. Otros son útiles como tareas de laboratorio que pudieran requerir dos o tres semanas de estudio e implementación. Algunos son proyectos de fin de curso desafiantes.

30.21 *(Análisis de textos)* La disponibilidad de computadoras con capacidades de manipulación de cadenas ha dado como resultado algunos métodos interesantes para analizar los escritos de grandes autores. Se ha dado mucha importancia para saber si realmente vivió William Shakespeare. Algunos estudiosos creen que existe una gran evidencia que indica que en realidad fue Cristopher Marlowe quien escribió las obras maestras que se atribuyen a Shakespeare. Los investigadores han utilizado computadoras para buscar similitudes en los escritos de estos dos autores. En este ejercicio se examinan tres métodos para analizar textos mediante una computadora.

a) Escriba una aplicación que lea una línea de texto desde el teclado e imprima una tabla que indique el número de ocurrencias de cada letra del alfabeto en el texto. Por ejemplo, la frase:

```
Ser o no ser: ése es el dilema:
```

contiene una "a", ninguna "b", ninguna "c", etcétera.

b) Escriba una aplicación que lea una línea de texto e imprima una tabla que indique el número de palabras de una letra, de dos letras, de tres letras, etcétera, que aparezcan en el texto. Por ejemplo, en la figura 30.25 se muestra la cuenta para la frase:

```
¿Qué es más noble para el espíritu?
```

c) Escriba una aplicación que lea una línea de texto e imprima una tabla que indique el número de ocurrencias de cada palabra distinta en el texto. La primera versión de su programa debe incluir las palabras en la tabla, en el mismo orden en el cual aparecen en el texto. Por ejemplo, las líneas:

```
Ser o no ser: ése es el dilema:
¿Qué es más noble para el espíritu?
```

Longitud de palabra	Ocurrencias
1	0
2	2
3	2
4	1
5	1
6	0
7	0
8	1

Figura 30.25 | La cuenta de longitudes de palabras para la cadena `"¿Qué es más noble para el espíritu?"`.

contiene la palabra “ser” dos veces, La palabra “o” una vez, la palabra “ése” una vez, etcétera. Una muestra más interesante (y útil) podría ser intentar con las palabras ordenadas alfabéticamente.

30.22 *(Impresión de fechas en varios formatos)* Las fechas se imprimen en varios formatos comunes. Dos de los formatos más utilizados son:

```
04/25/1955 y Abril 25, 1955
```

Escriba una aplicación que lea una fecha en el primer formato e imprima dicha fecha en el segundo formato.

30.23 *(Protección de cheques)* Las computadoras se utilizan con frecuencia en los sistemas de escritura de cheques, tales como aplicaciones para nóminas y para cuentas por pagar. Existen muchas historias extrañas acerca de cheques de nómina que se imprimen (por error) con montos que se exceden por millones. Los sistemas de emisión de cheques computarizados imprimen cantidades incorrectas debido al error humano o a una falla de la máquina. Los diseñadores de sistemas construyen controles en sus sistemas para evitar la emisión de dichos cheques erróneos.

Otro problema grave es la alteración intencional del monto de un cheque por alguien que planee cobrar un cheque de manera fraudulenta. Para evitar la alteración de un monto, la mayoría de los sistemas computarizados que emiten cheques emplean una técnica llamada protección de cheques. Los cheques diseñados para impresión por computadora contienen un número fijo de espacios en los cuales la computadora puede imprimir un monto. Suponga que un cheque contiene ocho espacios en blanco en los cuales la computadora puede imprimir el monto de un cheque de nómina semanal. Si el monto es grande, entonces se llenarán los ocho espacios. Por ejemplo:

```
1,230.60 (monto del cheque)
--------
12345678 (números de posición)
```

Por otra parte, si el monto es menor de $1,000, entonces varios espacios quedarían vacíos. Por ejemplo:

```
   99.87
--------
12345678
```

contiene tres espacios en blanco. Si se imprime un cheque con espacios en blanco, es más fácil para alguien alterar el monto del cheque. Para evitar que se altere el cheque, muchos sistemas de escritura de cheques insertan *asteriscos al principio* para proteger la cantidad, como se muestra a continuación:

```
***99.87
--------
12345678
```

Escriba una aplicación que reciba como entrada un monto a imprimir sobre un cheque y que lo escriba mediante el formato de protección de cheques, con asteriscos al principio si es necesario. Suponga que existen nueve espacios disponibles para imprimir el monto.

30.24 *(Escritura en letras del código de un cheque)* Para continuar con la discusión del ejercicio 30.23, reiteramos la importancia de diseñar sistemas de escritura de cheques para evitar la alteración de los montos de los cheques. Un método común de seguridad requiere que el monto del cheque se escriba tanto en números como en letras. Aun cuando alguien pueda alterar el monto numérico del cheque, es extremadamente difícil modificar el monto en letras. Escriba una aplicación que reciba como entrada un monto numérico para el cheque, y que escriba el equivalente del monto en letras. Por ejemplo, el monto 112.43 debe escribirse como

```
CIENTO DOCE CON 43/100
```

30.25 *(Clave Morse)* Quizá el más famoso de todos los esquemas de codificación es el código Morse, desarrollado por Samuel Morse en 1832 para usarlo con el sistema telegráfico. El código Morse asigna una serie de puntos y guiones a cada letra del alfabeto, cada dígito y algunos caracteres especiales (tales como el punto, la coma, los dos puntos y el punto y coma). En los sistemas orientados a sonidos, el punto representa un sonido corto y el guión representa un sonido largo. Otras representaciones de puntos y guiones se utilizan en los sistemas orientados a luces y sistemas de señalización con banderas. La separación entre palabras se indica mediante un espacio o, simplemente, con la ausencia de un punto o un guión. En un sistema orientado a sonidos, un espacio se indica por un tiempo breve durante el cual no se transmite sonido alguno. La versión internacional del código Morse aparece en la figura 30.26.

Escriba una aplicación que lea una frase en español y que codifique la frase en clave Morse. Además, escriba una aplicación que lea una frase en código Morse y que la convierta en su equivalente en español. Use un espacio en blanco entre cada letra en clave Morse, y tres espacios en blanco entre cada palabra en clave Morse.

Carácter	Código	Carácter	Código
A	.-	T	-
B	-...	U	..-
C	-.-.	V	...-
D	-..	W	.--
E	.	X	-..-
F	..-.	Y	-.--
G	--.	Z	--..
H	....		
I	..	*Dígitos*	
J	.---	1	.----
K	-.-	2	..---
L	.-..	3	...--
M	--	4	....-
N	-.	5	.....
O	---	6	-....
P	.--.	7	--...
Q	--.-	8	---..
R	.-.	9	----.
S	...	0	-----

Figura 30.26 | Las letras del alfabeto expresadas en código Morse internacional.

30.26 *(Aplicación de conversión al sistema métrico)* Escriba una aplicación que ayude al usuario a realizar conversiones métricas. Su aplicación debe permitir al usuario especificar los nombres de las unidades como cadenas (es decir, centímetros, litros, gramos, etcétera, para el sistema métrico, y pulgadas, cuartos, libras, etcétera, para el sistema inglés) y debe responder a preguntas simples tales como:

```
“¿Cuántas pulgadas hay en 2 metros?”
“¿Cuántos litros hay en 10 cuartos?”
```

Su programa debe reconocer conversiones inválidas. Por ejemplo, la pregunta:

```
“¿Cuántos pies hay en 5 kilogramos?”
```

no es correcta, debido a que los `"pies"` son unidades de longitud, mientras que los `"kilogramos"` son unidades de masa.

Sección especial: proyectos desafiantes de manipulación de cadenas

30.27 *(Proyecto: un corrector ortográfico)* Muchos paquetes populares de software de procesamiento de palabras cuentan con correctores ortográficos integrados. En este proyecto usted debe desarrollar su propia herramienta de corrección ortográfica. Le haremos unas sugerencias para ayudarlo a empezar. Sería conveniente que después le agregara más características. Use un diccionario computarizado (si tiene acceso a uno) como fuente de palabras.

¿Por qué escribimos tantas palabras en forma incorrecta? En algunos casos es porque simplemente no conocemos la manera correcta de escribirlas, por lo que tratamos de adivinar lo mejor que podemos. En otros casos, es porque transponemos dos letras (por ejemplo, "perdeterminado" en lugar de "predeterminado"). Algunas veces escribimos una letra doble por accidente (por ejemplo, "úttil" en vez de "útil"). Otras veces escribimos una tecla que está cerca de la que pretendíamos escribir (por ejemplo, "cunpleaños" en vez de "cumpleaños"), etcétera.

Diseñe e implemente una aplicación de corrección ortográfica en Java. Su aplicación debe mantener un arreglo de cadenas llamado `listaDePalabras`. Permita al usuario introducir estas cadenas. [*Nota:* en el capítulo 14 presentamos el procesamiento de archivos. Con esta capacidad, puede obtener las palabras para el corrector ortográfico de un diccionario computarizado almacenado en un archivo].

Su aplicación debe pedir al usuario que introduzca una palabra. La aplicación debe entonces buscar esa palabra en el arreglo `listaDePalabras`. Si la palabra se encuentra en el arreglo, su aplicación deberá imprimir `"La palabra está escrita correctamente"`. Si la palabra no se encuentra en el arreglo, su aplicación debe imprimir `"La palabra no está escrita correctamente"`. Después su aplicación debe tratar de localizar otras palabras en la `listaDePalabras` que puedan ser la palabra que el usuario trataba de escribir. Por ejemplo, puede probar con todas las transposiciones simples posibles de letras adyacentes para descubrir que la palabra "predeterminado" concuerda directamente con una palabra en `listaDePalabras`. Desde luego que esto implica que su programa comprobará todas las otras transposiciones posibles, como "rpedeterminado", "perdeterminado", "predetreminado", "predetemrinado" y "predetermniado". Cuando encuentre una nueva palabra que concuerde con una en la `listaDePalabras`, imprima esa palabra en un mensaje como

```
"¿Quiso decir "predeterminado"?".
```

Lleve a cabo otras pruebas, como reemplazar cada letra doble con una sola letra y cualquier otra prueba que pueda desarrollar para aumentar el valor de su corrector ortográfico.

30.28 *(Proyecto: un generador de crucigramas)* La mayoría de las personas han resuelto crucigramas, pero pocos han intentado generar uno. Aquí lo sugerimos como un proyecto de manipulación de cadenas que requiere una cantidad considerable de sofisticación y esfuerzo.

Hay muchas cuestiones que el programador tiene que resolver para hacer que funcione incluso hasta la aplicación generador de crucigramas más simple. Por ejemplo, ¿cómo representaría la cuadrícula de un crucigrama dentro de la computadora? ¿Debería utilizar una serie de cadenas o arreglos bidimensionales?

El programador necesita una fuente de palabras (es decir, un diccionario computarizado) a la que la aplicación pueda hacer referencia de manera directa. ¿De qué manera deben almacenarse estas palabras para facilitar las manipulaciones complejas que requiere la aplicación?

Si usted es realmente ambicioso, querrá generar la porción de "claves" del crucigrama, en la que se imprimen pistas breves para cada palabra "horizontal" y cada palabra "vertical". La sola impresión de la versión del crucigrama en blanco no es una tarea fácil.



Tabla de precedencia de los operadores

A.1 Precedencia de operadores

Los operadores se muestran en orden decreciente de precedencia, de arriba hacia abajo (figura A.1).

Operador	Descripción	Asociatividad
++	unario de postincremento	de derecha a izquierda
--	unario de postdecremento	
++	unario de preincremento	de derecha a izquierda
--	unario de predecremento	
+	unario de suma	
-	unario de resta	
!	unario de negación lógica	
~	unario de complemento a nivel de bits	
(*tipo*)	unario de conversión	
*	multiplicación	de izquierda a derecha
/	división	
%	residuo	
+	suma o concatenación de cadenas	de izquierda a derecha
-	resta	
<<	desplazamiento a la izquierda	de izquierda a derecha
>>	desplazamiento a la derecha con signo	
>>>	desplazamiento a la derecha sin signo	
<	menor que	
<=	menor o igual que	
>	mayor que	
>=	mayor o igual que	
instanceof	comparación de tipos	

Figura A.1 | Tabla de precedencia de los operadores. (Parte 1 de 2).

<table>
<tr><th>Operador</th><th>Descripción</th><th>Asociatividad</th></tr>
<tr><td>==
!=</td><td>es igual a
no es igual a</td><td>de izquierda a derecha</td></tr>
<tr><td>&</td><td>AND a nivel de bits
AND lógico booleano</td><td>de izquierda a derecha</td></tr>
<tr><td>^</td><td>OR excluyente a nivel de bits
OR excluyente lógico booleano</td><td>de izquierda a derecha</td></tr>
<tr><td>|</td><td>OR incluyente a nivel de bits
OR incluyente lógico booleano</td><td>de izquierda a derecha</td></tr>
<tr><td>&&</td><td>AND condicional</td><td>de izquierda a derecha</td></tr>
<tr><td>||</td><td>OR condicional</td><td>de izquierda a derecha</td></tr>
<tr><td>?:</td><td>condicional</td><td>de derecha a izquierda</td></tr>
<tr><td>=
+=
-=
*=
/=
%=
&=
^=
|=
<<=
>>=
>>>=</td><td>asignación
asignación, suma
asignación, resta
asignación, multiplicación
asignación, división
asignación, residuo
asignación, AND a nivel de bits
asignación, OR excluyente a nivel de bits
asignación, OR incluyente a nivel de bits
asignación, desplazamiento a la izquierda a nivel de bits
asignación, desplazamiento a la derecha a nivel de bits con signo
asignación, desplazamiento a la derecha a nivel de bits sin signo</td><td>de derecha a izquierda</td></tr>
</table>

Figura A.1 | Tabla de precedencia de los operadores. (Parte 2 de 2).

B

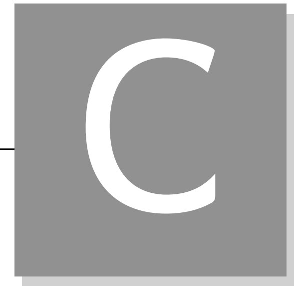
Conjunto de caracteres ASCII

	0	1	2	3	4	5	6	7	8	9
0	nul	soh	stx	ext	eot	enq	ack	bel	bs	ht
1	nl	vt	ff	cr	so	si	dle	dc1	dc2	dc3
2	dc4	nak	syn	etb	can	em	sub	esc	fs	gs
3	rs	us	sp	!	"	#	$	%	&	‘
4	(	)	*	+	,	-	.	/	0	1
5	2	3	4	5	6	7	8	9	:	;
6	<	=	>	?	@	A	B	C	D	E
7	F	G	H	I	J	K	L	M	N	O
8	P	Q	R	S	T	U	V	W	X	Y
9	Z	[	\	]	^	_	’	a	b	c
10	d	e	f	g	h	i	j	k	l	m
11	n	o	p	q	r	s	t	u	v	w
12	x	y	z	{	|	}	~	del		

Figura B.1 | El conjunto de caracteres ASCII.

Los dígitos a la izquierda de la tabla son los dígitos izquierdos del equivalente decimal (0-127) del código de caracteres, y los dígitos en la parte superior de la tabla son los dígitos derechos del código de caracteres. Por ejemplo, el código de carácter para la “F” es 70, mientras que para el “&” es 38.

La mayoría de los usuarios de este libro estarán interesados en el conjunto de caracteres ASCII utilizado para representar los caracteres del idioma español en muchas computadoras. El conjunto de caracteres ASCII es un subconjunto del conjunto de caracteres Unicode utilizado por Java para representar caracteres de la mayoría de los lenguajes existentes en el mundo. Para obtener más información acerca del conjunto de caracteres Unicode, vea el apéndice I, Unicode®, que se incluye como bono Web.



Palabras clave y palabras reservadas

Palabras clave en Java				
`abstract`	`assert`	`boolean`	`break`	`byte`
`case`	`catch`	`char`	`class`	`continue`
`default`	`do`	`double`	`else`	`enum`
`extends`	`final`	`finally`	`float`	`for`
`if`	`implements`	`import`	`instanceof`	`int`
`interface`	`long`	`native`	`new`	`package`
`private`	`protected`	`public`	`return`	`short`
`static`	`strictfp`	`super`	`switch`	`synchronized`
`this`	`throw`	`throws`	`transient`	`try`
`void`	`volatile`	`while`		
Palabras clave que no se utilizan actualmente				
`const`	`goto`			

Figura C.1 | Palabras clave de Java.

Java también contiene las palabras reservadas `true` y `false`, las cuales son literales `bolean`, y `null`, que es la literal que representa una referencia a nada. Al igual que las palabras clave, esas palabras reservadas no se pueden utilizar como identificadores.



Tipos primitivos

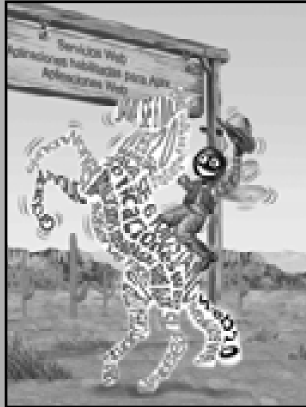
Tipo	Tamaño en bits	Valores	Estándar
`boolean`		`true o false`	
[*Nota:* una representación `bolean` es específica para la Máquina virtual de Java en cada plataforma].			
`char`	16	`'\u0000 '` a `'\uFFFF '` (0 a 65535)	(ISO, conjunto de caracteres Unicode)
`byte`	8	–128 a +127 ($–2^7$ a $2^7 – 1$)	
`short`	16	–32,768 a +32,767 ($–2^{15}$ a $2^{15} – 1$)	
`int`	32	–2,147,483,648 a +2,147,483,647 ($–2^{31}$ a $2^{31} – 1$)	
`long`	64	–9,223,372,036,854,775,808 a +9,223,372,036,854,775,807 ($–2^{63}$ a $2^{63} – 1$)	
`float`	32	*Rango negativo:* –3.4028234663852886E+38 a –1.40129846432481707e–45 *Rango positivo:* 1.40129846432481707e–45 a 3.4028234663852886E+38	(IEEE 754, punto flotante)
`double`	64	*Rango negativo:* –1.7976931348623157E+308 a –4.94065645841246544e–324 *Rango positivo:* 4.94065645841246544e–324 a 1.7976931348623157E+308	(IEEE 754, punto flotante)

Figura D.1 | Tipos primitivos de Java.

Para obtener más información acerca de IEEE 754, visite `grouper.ieee.org/groups/754/`. Para obtener más información sobre Unicode, vea el apéndice I, Unicode®.

E

Sistemas numéricos



He aquí sólo los números ratificados.

—William Shakespeare

La naturaleza tiene un cierto tipo de sistema de coordenadas aritméticas-geométricas, ya que cuenta con todo tipo de modelos. Lo que experimentamos de la naturaleza está en los modelos, y todos los modelos de la naturaleza son tan bellos. Se me ocurrió que el sistema de la naturaleza debe ser una verdadera belleza, porque en la química encontramos que las asociaciones se encuentran siempre en hermosos números enteros; no hay fracciones.

—Richard Buckminster Fuller

OBJETIVOS

En este apéndice aprenderá a:

- Comprender los conceptos acerca de los sistemas numéricos como base, valor posicional y valor simbólico.
- Trabajar con los números representados en los sistemas numéricos binario, octal y hexadecimal.
- Abreviar los números binarios como octales o hexadecimales.
- Convertir los números octales y hexadecimales en binarios.
- Realizar conversiones hacia y desde números decimales y sus equivalentes en binario, octal y hexadecimal.
- Comprender el funcionamiento de la aritmética binaria y la manera en que se representan los números binarios negativos, utilizando la notación de complemento a dos.

Plan general

E.1 Introducción

En este apéndice presentaremos los sistemas numéricos clave que utilizan los programadores de Java, especialmente cuando trabajan en proyectos de software que requieren de una estrecha interacción con el hardware a nivel de máquina. Entre los proyectos de este tipo están los sistemas operativos, el software de redes computacionales, los compiladores, sistemas de bases de datos y aplicaciones que requieren de un alto rendimiento.

Cuando escribimos un entero, como 227 o –63, en un programa de Java, se asume que el número está en el sistema numérico decimal (base 10). Los dígitos en el sistema numérico decimal son 0, 1, 2, 3, 4, 5, 6, 7, 8 y 9. El dígito más bajo es el 0 y el más alto es el 9 (uno menos que la base, 10). En su interior, las computadoras utilizan el sistema numérico binario (base 2). Este sistema numérico sólo tiene dos dígitos: 0 y 1. El dígito más bajo es el 0 y el más alto es el 1 (uno menos que la base, 2).

Como veremos, los números binarios tienden a ser mucho más extensos que sus equivalentes decimales. Los programadores que trabajan con lenguajes ensambladores y en lenguajes de alto nivel como Java, que les permiten llegar hasta el nivel de máquina, encuentran que es complicado trabajar con números binarios. Por eso existen otros dos sistemas numéricos, el sistema numérico octal (base 8) y el sistema numérico hexadecimal (base 16), que son populares debido a que permiten abreviar los números binarios de una manera conveniente.

En el sistema numérico octal, los dígitos utilizados son del 0 al 7. Debido a que tanto el sistema numérico binario como el octal tienen menos dígitos que el sistema numérico decimal, sus dígitos son los mismos que sus correspondientes en decimal.

El sistema numérico hexadecimal presenta un problema, ya que requiere de 16 dígitos: el dígito más bajo es 0 y el más alto tiene un valor equivalente al 15 decimal (uno menos que la base, 16). Por convención utilizamos las letras de la A a la F para representar los dígitos hexadecimales que corresponden a los valores decimales del 10 al 15. Por lo tanto, en hexadecimal podemos tener números como el 876, que consisten solamente de dígitos similares a los decimales; números como 8A55F que consisten de dígitos y letras; y números como FFE que consisten solamente de letras. En ocasiones un número hexadecimal puede coincidir con una palabra común como FACE o FEED (en inglés); esto puede parecer extraño para los programadores acostumbrados a trabajar con números. Los dígitos de los sistemas numéricos binario, octal, decimal y hexadecimal se sintetizan en las figuras E.1 y E.2.

Cada uno de estos sistemas numéricos utilizan la notación posicional: cada posición en la que se escribe un dígito tiene un valor posicional distinto. Por ejemplo, en el número decimal 937 (el 9, el 3 y el 7 se conocen como valores simbólicos) decimos que el 7 se escribe en la posición de las unidades; el 3, en la de las decenas; y el 9, en la de las centenas. Observe que cada una de estas posiciones es una potencia de la base (10) y que estas potencias empiezan en 0 y aumentan de 1 en 1 a medida que nos desplazamos hacia la izquierda por el número (figura E.3).

Para números decimales más extensos, las siguientes posiciones a la izquierda serían: de millares (10 a la tercera potencia), de decenas de millares (10 a la cuarta potencia), de centenas de millares (10 a la quinta potencia), de los millones (10 a la sexta potencia), de decenas de millones (10 a la séptima potencia), y así sucesivamente.

En el número binario 101 decimos que el 1 más a la derecha se escribe en la posición de los unos, el 0 se escribe en la posición de los dos y el 1 de más a la izquierda se escribe en la posición de los cuatros. Observe que cada una de estas posiciones es una potencia de la base (2) y que estas potencias empiezan en 0 y aumentan de 1 en 1 a medida que nos desplazamos hacia la izquierda por el número (figura E.4). Por lo tanto, $101 = 2^2 + 2^0 = 4 + 1 = 5$.

Para números binarios más extensos, las siguientes posiciones a la izquierda serían la posición de los ochos (2 a la tercera potencia), la posición de los dieciséis (2 a la cuarta potencia), la posición de los treinta y dos (2 a la quinta potencia), la posición de los sesenta y cuatros (2 a la sexta potencia), y así sucesivamente.

Dígito binario	Dígito octal	Dígito decimal	Dígito hexadecimal
0	0	0	0
1	1	1	1
	2	2	2
	3	3	3
	4	4	4
	5	5	5
	6	6	6
	7	7	7
		8	8
		9	9
			A (valor de 10 en decimal)
			B (valor de 11 en decimal)
			C (valor de 12 en decimal)
			D (valor de 13 en decimal)
			E (valor de 14 en decimal)
			F (valor de 15 en decimal)

Figura E.1 | Dígitos de los sistemas numéricos binario, octal, decimal y hexadecimal.

Atributo	Binario	Octal	Decimal	Hexadecimal
Base	2	8	10	16
Dígito más bajo	0	0	0	0
Dígito más alto	1	7	9	F

Figura E.2 | Comparación de los sistemas binario, octal, decimal y hexadecimal.

Valores posicionales en el sistema numérico decimal			
Dígito decimal	9	3	7
Nombre de la posición	Centenas	Decenas	Unidades
Valor posicional	100	10	1
Valor posicional como potencia de la base (10)	10^2	10^1	10^0

Figura E.3 | Valores posicionales en el sistema numérico decimal.

En el número octal 425, decimos que el 5 se escribe en la posición de los unos, el 2 se escribe en la posición de los ochos y el 4 se escribe en la posición de los sesenta y cuatros. Observe que cada una de estas posiciones es una potencia de la base (8) y que estas potencias empiezan en 0 y aumentan de 1 en 1 a medida que nos desplazamos hacia la izquierda por el número (figura E.5).

Para números octales más extensos, las siguientes posiciones a la izquierda sería la posición de los quinientos doces (8 a la tercera potencia), la posición de los cuatro mil noventa y seis (8 a la cuarta potencia), la posición de los treinta y dos mil setecientos sesenta y ochos (8 a la quinta potencia), y así sucesivamente.

Valores posicionales en el sistema numérico binario			
Dígito binario	1	0	1
Nombre de la posición	Cuatro	Dos	Unos
Valor posicional	4	2	1
Valor posicional como potencia de la base (2)	2^2	2^1	2^0

Figura E.4 | Valores posicionales en el sistema numérico binario.

Valores posicionales en el sistema numérico octal			
Dígito octal	4	2	5
Nombre de la posición	Sesenta y cuatros	Ochos	Unos
Valor posicional	64	8	1
Valor posicional como potencia de la base (8)	8^2	8^1	8^0

Figura E.5 | Valores posicionales en el sistema numérico octal.

Valores posicionales en el sistema numérico hexadecimal			
Dígito hexadecimal	3	D	A
Nombre de la posición	Doscientos cincuenta y seis	Dieciséis	Unos
Valor posicional	256	16	1
Valor posicional como potencia de la base (16)	16^2	16^1	16^0

Figura E.6 | Valores posicionales en el sistema numérico hexadecimal.

En el número hexadecimal 3DA, decimos que la A se escribe en la posición de los unos, la D se escribe en la posición de los dieciséis y el 3 se escribe en la posición de los doscientos cincuenta y seis. Observe que cada una de estas posiciones es una potencia de la base (16) y que estas potencias empiezan en 0 y aumentan de 1 en 1 a medida que nos desplazamos hacia la izquierda por el número (figura E.6).

Para números hexadecimales más extensos, las siguientes posiciones a la izquierda serían la posición de los cuatro mil noventa y seis (16 a la tercera potencia), la posición de los sesenta y cinco mil quinientos treinta y seis (16 a la cuarta potencia), y así sucesivamente.

E.2 Abreviatura de los números binarios como números octales y hexadecimales

En computación, el uso principal de los números octales y hexadecimales es para abreviar representaciones binarias demasiado extensas. La figura E.7 muestra que los números binarios extensos pueden expresarse más concisamente en sistemas numéricos con bases mayores que en el sistema numérico binario.

Una relación especialmente importante que tienen tanto el sistema numérico octal como el hexadecimal con el sistema binario es que las bases de los sistemas octal y hexadecimal (8 y 16, respectivamente) son potencias de la base del sistema numérico binario (base 2). Considere el siguiente número binario de 12 dígitos y sus equivalentes en octal y hexadecimal. Vea si puede determinar cómo esta relación hace que sea conveniente el abreviar los números binarios en octal o hexadecimal. La respuesta sigue después de los números.

Número decimal	Representación binaria	Representación octal	Representación hexadecimal
0	0	0	0
1	1	1	1
2	10	2	2
3	11	3	3
4	10	4	4
5	101	5	5
6	110	6	6
7	111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F
16	10000	20	10

Figura E.7 | Equivalentes en decimal, binario, octal y hexadecimal.

Número binario	Equivalente en octal	Equivalente en hexadecimal
100011010001	4321	8D1

Para convertir fácilmente el número binario en octal, sólo divida el número binario de 12 dígitos en grupos de tres bits consecutivos y escriba esos grupos por encima de los dígitos correspondientes del número octal, como se muestra a continuación:

```
100   011   010   001
4     3     2     1
```

Observe que el dígito octal que escribió debajo de cada grupo de tres bits corresponde precisamente al equivalente octal de ese número binario de 3 dígitos que se muestra en la figura E.7.

El mismo tipo de relación puede observarse al convertir números de binario a hexadecimal. Divida el número binario de 12 dígitos en grupos de cuatro bits consecutivos y escriba esos grupos por encima de los dígitos correspondientes del número hexadecimal, como se muestra a continuación:

```
1000   1101   0001
8      D      1
```

Observe que el dígito hexadecimal que escribió debajo de cada grupo de cuatro bits corresponde precisamente al equivalente hexadecimal de ese número binario de 4 dígitos que se muestra en la figura E.7.

E.3 Conversión de números octales y hexadecimales a binarios

En la sección anterior vimos cómo convertir números binarios a sus equivalentes en octal y hexadecimal, formando grupos de dígitos binarios y simplemente volviéndolos a escribir como sus valores equivalentes en dígitos octales o hexadecimales. Este proceso puede utilizarse en forma inversa para producir el equivalente en binario de un número octal o hexadecimal.

Por ejemplo, el número octal 653 se convierte en binario simplemente escribiendo el 6 como su equivalente binario de 3 dígitos 110, el 5 como su equivalente binario de 3 dígitos 101 y el 3 como su equivalente binario de 3 dígitos 011 para formar el número binario de 9 dígitos 110101011.

El número hexadecimal FAD5 se convierte en binario simplemente escribiendo la F como su equivalente binario de 4 dígitos 1111, la A como su equivalente binario de 4 dígitos 1010, la D como su equivalente binario de 4 dígitos 1101 y el 5 como su equivalente binario de 4 dígitos 0101, para formar el número binario de 16 dígitos 1111101011010101.

E.4 Conversión de un número binario, octal o hexadecimal a decimal

Como estamos acostumbrados a trabajar con el sistema decimal, a menudo es conveniente convertir un número binario, octal o hexadecimal en decimal para tener una idea de lo que "realmente" vale el número. Nuestros diagramas en la sección E.1 expresan los valores posicionales en decimal. Para convertir un número en decimal desde otra base, multiplique el equivalente en decimal de cada dígito por su valor posicional y sume estos productos. Por ejemplo, el número binario 110101 se convierte en el número 53 decimal, como se muestra en la figura E.8.

Para convertir el número 7614 octal en el número 3980 decimal utilizamos la misma técnica, esta vez utilizando los valores posicionales apropiados para el sistema octal, como se muestra en la figura E.9.

Para convertir el número AD3B hexadecimal en el número 44347 decimal utilizamos la misma técnica, esta vez empleando los valores posicionales apropiados para el sistema hexadecimal, como se muestra en la figura E.10.

Conversión de un número binario en decimal						
Valores posicionales:	32	16	8	4	2	1
Valores simbólicos:	1	1	0	1	0	1
Productos:	1*32=32	1*16=16	0*8=0	1*4=4	0*2=0	1*1=1
Suma:	= 32 + 16 + 0 + 4 + 0s + 1 = 53					

Figura E.8 | Conversión de un número binario en decimal.

Conversión de un número octal en decimal				
Valores posicionales:	512	16	32	58
Valores simbólicos:	7	32	32	32
Productos:	7*512=3584	6*64=384	1*8=8	4*1=4
Suma:	= 3584 + 384 + 8 + 4 = 3980			

Figura E.9 | Conversión de un número octal en decimal.

Conversión de un número hexadecima; en decimal				
Valores posicionales:	4096	256	16	1
Valores simbólicos:	A	D	3	B
Productos:	A*4096=40960	D*256=3328	3*16=48	B*1=11
Suma:	= 40960 + 3328 + 48 + 11 = 44347			

Figura E.10 | Conversión de un número hexadecimal en decimal.

E.5 Conversión de un número decimal a binario, octal o hexadecimal

Las conversiones de la sección E.4 siguen naturalmente las convenciones de la notación posicional. Las conversiones de decimal a binario, octal o hexadecimal también siguen estas convenciones.

Suponga que queremos convertir el número 57 decimal en binario. Empezamos escribiendo los valores posicionales de las columnas de derecha a izquierda, hasta llegar a una columna cuyo valor posicional sea mayor que el número decimal. Como no necesitamos esa columna, podemos descartarla. Por lo tanto, primero escribimos:

Valores posicionales:	64	32	16	8	4	2	1

Luego descartamos la columna con el valor posicional de 64, dejando:

Valores posicionales:	32	16	8	4	2	1

A continuación, empezamos a trabajar desde la columna más a la izquierda y nos vamos desplazando hacia la derecha. Dividimos 57 entre 32 y observamos que hay un 32 en 57, con un residuo de 25, por lo que escribimos 1 en la columna de los 32. Dividimos 25 entre 16 y observamos que hay un 16 en 25, con un residuo de 9, por lo que escribimos 1 en la columna de los 16. Dividimos 9 entre 8 y observamos que hay un 8 en 9 con un residuo de 1. Las siguientes dos columnas producen el cociente de cero cuando se divide 1 entre sus valores posicionales, por lo que escribimos 0 en las columnas de los 4 y de los 2. Por último, 1 entre 1 es 1, por lo que escribimos 1 en la columna de los 1. Esto nos da:

Valores posicionales:	32	16	8	4	2	1
Valores simbólicos:	1	1	1	0	0	1

y, por lo tanto, el 57 decimal es equivalente al 111001 binario.

Para convertir el número decimal 103 en octal, empezamos por escribir los valores posicionales de las columnas hasta llegar a una columna cuyo valor posicional sea mayor que el número decimal. Como no necesitamos esa columna, podemos descartarla. Por lo tanto, primero escribimos:

Valores posicionales:	512	64	8	1

Luego descartamos la columna con el valor posicional de 512, lo que nos da:

Valores posicionales:	64	8	1

A continuación, empezamos a trabajar desde la columna más a la izquierda y nos vamos desplazando hacia la derecha. Dividimos 103 entre 64 y observamos que hay un 64 en 103 con un residuo de 39, por lo que escribimos 1 en la columna de los 64. Dividimos 39 entre 8 y observamos que el 8 cabe cuatro veces en 39 con un residuo de 7, por lo que escribimos 4 en la columna de los 8. Por último, dividimos 7 entre 1 y observamos que el 1 cabe siete veces en 7 y no hay residuo, por lo que escribimos 7 en la columna de los 1. Esto nos da:

Valores posicionales:	64	8	1
Valores simbólicos:	1	4	7

y por lo tanto, el 103 decimal es equivalente al 147 octal.

Para convertir el número decimal 375 en hexadecimal, empezamos por escribir los valores posicionales de las columnas hasta llegar a una columna cuyo valor posicional sea mayor que el número decimal. Como no necesitamos esa columna, podemos descartarla. Por consecuencia, primero escribimos:

Valores posicionales:	4096	256	16	1

Luego descartamos la columna con el valor posicional de 4096, lo que nos da:

Valores posicionales:	256	16	1

A continuación, empezamos a trabajar desde la columna más a la izquierda y nos vamos desplazando hacia la derecha. Dividimos 375 entre 256 y observamos que 256 cabe una vez en 375 con un residuo de 119, por lo que escribimos 1 en la columna de los 256. Dividimos 119 entre 16 y observamos que el 16 cabe siete veces en 119 con un residuo de 7, por lo que escribimos 7 en la columna de los 16. Por último, dividimos 7 entre 1 y

observamos que el 1 cabe siete veces en 7 y no hay residuo, por lo que escribimos 7 en la columna de los 1. Esto produce:

Valores posicionales:	256	16	1
Valores simbólicos:	1	7	7

y, por lo tanto, el 375 decimal es equivalente al 177 hexadecimal.

E.6 Números binarios negativos: notación de complemento a dos

La discusión en este apéndice se ha enfocado hasta ahora en números positivos. En esta sección explicaremos cómo las computadoras representan números negativos mediante el uso de la notación de *complementos a dos*. Primero explicaremos cómo se forma el complemento a dos de un número binario y después mostraremos por qué representa el valor negativo de dicho número binario.

Considere una máquina con enteros de 32 bits. Suponga que se ejecuta la siguiente instrucción:

```
int valor = 13;
```

La representación en 32 bits de `valor` es:

```
00000000 00000000 00000000 00001101
```

Para formar el negativo de `valor`, primero formamos su *complemento a uno* aplicando el operador de complemento a nivel de bits de Java (~):

```
complementoAUnoDeValor = ~valor;
```

Internamente, `~valor` es ahora valor con cada uno de sus bits invertidos; los unos se convierten en ceros y los ceros en unos, como se muestra a continuación:

```
valor:
00000000 00000000 00000000 00001101

~valor (es decir, el complemento a uno de valor):
11111111 11111111 11111111 11110010
```

Para formar el complemento a dos de valor, simplemente sumamos uno al complemento a uno de valor. Por lo tanto:

```
El complemento a dos de valor es:
11111111 11111111 11111111 11110011
```

Ahora, si esto de hecho es igual a –13, deberíamos poder sumarlo al 13 binario y obtener como resultado 0. Comprobemos esto:

```
 00000000 00000000 00000000 00001101
+11111111 11111111 11111111 11110011
------------------------------------
 00000000 00000000 00000000 00000000
```

El bit de acarreo que sale de la columna que está más a la izquierda se descarta y evidentemente obtenemos 0 como resultado. Si sumamos el complemento a uno de un número a ese mismo número, todos los dígitos del resultado serían iguales a 1. La clave para obtener un resultado en el que todos los dígitos sean cero es que el complemento a dos es 1 más que el complemento a 1. La suma de 1 hace que el resultado de cada columna sea 0 y se acarrea un 1. El acarreo sigue desplazándose hacia la izquierda hasta que se descarta en el bit que está más a la izquierda, con lo que todos los dígitos del número resultante son iguales a cero.

En realidad, las computadoras realizan una suma como:

```
x = a - valor;
```

mediante la suma del complemento a dos de `valor` con `a`, como se muestra a continuación:

```
x = a + (~valor + 1);
```

Suponga que `a` es 27 y que `valor` es 13 como en el ejemplo anterior. Si el complemento a dos de `valor` es en realidad el negativo de éste, entonces al sumar el complemento de dos de `valor` con `a` se produciría el resultado de 14. Comprobemos esto:

```
a (es decir, 27)     00000000 00000000 00000000 00011011
+(~valor + 1)       +11111111 11111111 11111111 11110011
                    -----------------------------------
                     00000000 00000000 00000000 00001110
```

lo que ciertamente da como resultado 14.

Resumen

- Cuando escribimos un entero como 19, 227 o –63, en un programa de Java, suponemos que el número se encuentra en el sistema numérico decimal (base 10). Los dígitos en el sistema numérico decimal son 0, 1, 2, 3, 4, 5, 6, 7, 8 y 9. El dígito más bajo es el 0 y el más alto es el 9 (uno menos que la base, 10).
- En su interior, las computadoras utilizan el sistema numérico binario (base 2). Este sistema numérico sólo tiene dos dígitos: 0 y 1. El dígito más bajo es el 0 y el más alto es el 1 (uno menos que la base, 2).
- El sistema numérico octal (base 8) y el sistema numérico hexadecimal (base 16) son populares debido a que permiten abreviar los números binarios de una manera conveniente.
- Los dígitos que se utilizan en el sistema numérico octal son del 0 al 7.
- El sistema numérico hexadecimal presenta un problema, ya que requiere de dieciséis dígitos: el dígito más bajo es 0 y el más alto tiene un valor equivalente al 15 decimal (uno menos que la base, 16). Por convención utilizamos las letras de la A a la F para representar los dígitos hexadecimales que corresponden a los valores decimales del 10 al 15.
- Cada uno de estos sistemas numéricos utilizan la notación posicional: cada posición en la que se escribe un dígito tiene un distinto valor posicional.
- Una relación especialmente importante que tienen tanto el sistema numérico octal como el hexadecimal con el sistema binario es que las bases de los sistemas octal y hexadecimal (8 y 16, respectivamente) son potencias de la base del sistema numérico binario (base 2).
- Para convertir un número octal en binario, sustituya cada dígito octal con su equivalente binario de tres dígitos.
- Para convertir un número hexadecimal en binario, simplemente sustituya cada dígito hexadecimal con su equivalente binario de cuatro dígitos.
- Como estamos acostumbrados a trabajar con el sistema decimal, es conveniente convertir un número binario, octal o hexadecimal en decimal para tener una idea de lo que "realmente" vale el número.
- Para convertir un número en decimal desde otra base, multiplique el equivalente en decimal de cada dígito por su valor posicional y sume estos productos.
- Las computadoras representan números negativos mediante el uso de la notación de complementos a dos.
- Para formar el negativo de un valor en binario, primero formamos su complemento a uno aplicando el operador de complemento a nivel de bits de Java (~). Esto invierte los bits del valor. Para formar el complemento a dos de un valor, simplemente sumamos uno al complemento a uno de ese valor.

Terminología

base
conversiones
dígito
notación de complementos a dos
notación de complementos a uno
notación posicional
operador de complemento a nivel de bits (~)
sistema numérico binario
sistema numérico de base 10
sistema numérico de base 16
sistema numérico de base 2
sistema numérico de base 8
sistema numérico decimal
sistema numérico hexadecimal
sistema numérico octal
valor negativo
valor posicional
valor simbólico

Ejercicios de autoevaluación

E.1 Las bases de los sistemas numéricos decimal, binario, octal y hexadecimal son ________________, ____________, ________________ y ________________, respectivamente.

E.2 En general, las representaciones en decimal, octal y hexadecimal de un número binario dado contienen (más/menos) dígitos de los que contiene el número binario.

E.3 (*Verdadero/falso*) Una de las razones populares de utilizar el sistema numérico decimal es que forma una notación conveniente para abreviar números binarios, en la que simplemente se sustituye un dígito decimal por cada grupo de cuatro dígitos binarios.

E.4 La representación (octal/hexadecimal/decimal) de un valor binario grande es la más concisa (de las alternativas dadas).

E.5 (*Verdadero/falso*) El dígito de mayor valor en cualquier base es uno más que la base.

E.6 (*Verdadero/falso*) El dígito de menor valor en cualquier base es uno menos que la base.

E.7 El valor posicional del dígito que se encuentra más a la derecha en cualquier número, ya sea binario, octal, decimal o hexadecimal es siempre ________________.

E.8 El valor posicional del dígito que está a la izquierda del dígito que se encuentra más a la derecha en cualquier número, ya sea binario, octal, decimal o hexadecimal es siempre igual a ________________.

E.9 Complete los valores que faltan en esta tabla de valores posicionales para las cuatro posiciones que están más a la derecha en cada uno de los sistemas numéricos indicados:

decimal	1000	100	10	1
hexadecimal	...	256	...	...
binario	...	...	...	...
octal	512	...	8	...

E.10 Convierta el número binario 110101011000 en octal y en hexadecimal.

E.11 Convierta el número hexadecimal FACE en binario.

E.12 Convierta el número octal 7316 en binario.

E.13 Convierta el número hexadecimal 4FEC en octal. (*Sugerencia:* primero convierta el número 4FEC en binario y después convierta el número resultante en octal).

E.14 Convierta el número binario 1101110 en decimal.

E.15 Convierta el número octal 317 en decimal.

E.16 Convierta el número hexadecimal EFD4 en decimal.

E.17 Convierta el número decimal 177 en binario, en octal y en hexadecimal.

E.18 Muestre la representación binaria del número decimal 417. Después muestre el complemento a uno de 417 y el complemento a dos del mismo número.

E.19 ¿Cuál es el resultado cuando se suma el complemento a dos de un número con ese mismo número?

Respuestas a los ejercicios de autoevaluación

E.1 10, 2, 8, 16.

E.2 Menos.

E.3 Falso. El hexadecimal hace esto.

E.4 Hexadecimal.

E.5 Falso. El dígito de mayor valor en cualquier base es uno menos que la base.

E.6 Falso. El dígito de menor valor en cualquier base es cero.

E.7 1 (La base elevada a la potencia de cero).

E.8 La base del sistema numérico.

E.9 Complete los valores que faltan en esta tabla de valores posicionales para las cuatro posiciones que están más a la derecha en cada uno de los sistemas numéricos indicados:

decimal	1000	100	10	1
hexadecimal	4096	256	16	1
binario	8	4	2	1
octal	512	64	8	1

E.10 6530 octal; D58 hexadecimal.

E.11 1111 1010 1100 1110 binario.

E.12 111 011 001 110 binario.

E.13 0 100 111 111 101 100 binario; 47754 octal.

E.14 2+4+8+32+64=110 decimal.

E.15 7+1*8+3*64=7+8+192=207 decimal.

E.16 4+13*16+15*256+14*4096=61396 decimal.

E.17 177 decimal
en binario:

```
256 128 64 32 16 8 4 2 1
128 64 32 16 8 4 2 1
(1*128)+(0*64)+(1*32)+(1*16)+(0*8)+(0*4)+(0*2)+(1*1)
10110001
```

en octal:

```
512 64 8 1
64 8 1
(2*64)+(6*8)+(1*1)
261
```

en hexadecimal:

```
256 16 1
16 1
(11*16)+(1*1)
(B*16)+(1*1)
B1
```

E.18 Binario:

```
512 256 128 64 32 16 8 4 2 1
256 128 64 32 16 8 4 2 1
(1*256)+(1*128)+(0*64)+(1*32)+(0*16)+(0*8)+(0*4)+(0*2)+
(1*1)
110100001
```

Complemento a uno: 001011110
Complemento a dos: 001011111
Comprobación: Número binario original + su complemento a dos:

```
110100001
001011111
---------
000000000
```

E.19 Cero.

Ejercicios

E.20 Algunas personas argumentan que muchos de nuestros cálculos se realizarían más fácilmente en el sistema numérico de base 12, ya que el 12 puede dividirse por muchos más números que el 10 (por la base 10). ¿Cuál es el dígito de menor valor en la base 12? ¿Cuál podría ser el símbolo con mayor valor para un dígito en la base 12? ¿Cuáles son los valores posicionales de las cuatro posiciones más a la derecha de cualquier número en el sistema numérico de base 12?

E.21 Complete la siguiente tabla de valores posicionales para las cuatro posiciones más a la derecha en cada uno de los sistemas numéricos indicados:

decimal	1000	100	10	1
base 6	...	...	6	...
base 13	...	169	...	...
base 3	27	...	...	...

E.22 Convierta el número binario 100101111010 en octal y en hexadecimal.

E.23 Convierta el número hexadecimal 3A7D en binario.

E.24 Convierta el número hexadecimal 765F en octal. (*Sugerencia:* primero conviértalo en binario y después convierta el número resultante en octal).

E.25 Convierta el número binario 1011110 en decimal.

E.26 Convierta el número octal 426 en decimal.

E.27 Convierta el número hexadecimal FFFF en decimal.

E.28 Convierta el número decimal 299 en binario, en octal y en hexadecimal.

E.29 Muestre la representación binaria del número decimal 779. Después muestre el complemento a uno de 779 y el complemento a dos del mismo número.

E.30 Muestre el complemento a dos del valor entero –1 en una máquina con enteros de 32 bits.

F

GroupLayout

F.1 Introducción

Java SE 6 incluye un nuevo y poderoso administrador de esquemas llamado **GroupLayout**, el cual es el administrador de esquemas predeterminado en el IDE Netbeans 5.5 (www.netbeans.org). En este apéndice veremos las generalidades acerca de GroupLayout, y después demostraremos cómo usar el **diseñador de GUI Matisse** del IDE Netbeans 5.5 para crear una GUI mediante el uso de GroupLayout para posicionar los componentes. NetBeans genera el código de GroupLayout por el programador de manera automática. Aunque podemos escribir código de GroupLayout en forma manual, en la mayoría de los casos es mejor utilizar una herramienta de diseño de GUI tal como la que proporciona Netbeans, para sacar provecho al poder de GroupLayout. Para obtener más detalles acerca de GroupLayout, consulte la lista de recursos Web al final de este apéndice.

F.2 Fundamentos de GroupLayout

En los capítulos 11 y 22 presentamos varios administradores de esquemas que proporcionan herramientas de esquemas de GUI. También vimos cómo combinar administradores de esquemas y varios contenedores para crear esquemas más complejos. La mayoría de los administradores de esquemas no nos proporcionan un control preciso sobre el posicionamiento de los componentes. En el capítulo 22 vimos GridBagLayout, que proporciona un control más preciso sobre la posición y el tamaño de los componentes de GUI del programador. Nos permite especificar la posición vertical y horizontal de cada componente, el número de filas y columnas que ocupa cada componente en la cuadrícula, y la forma en que los componentes aumentan y reducen su tamaño, a medida que cambia el tamaño del contenedor. Todo esto se especifica al mismo tiempo con un objeto GridBagConstraints. La clase GroupLayout es el siguiente paso en la administración de esquemas. GroupLayout es más flexible, ya que el programador puede especificar los esquemas horizontal y vertical de sus componentes de manera independiente.

Arreglos en serie y en paralelo

Los componentes se ordenan en secuencia o en paralelo. Los tres objetos JButton de la figura F.1 tienen una **orientación horizontal secuencial**: aparecen de izquierda a derecha en secuencia. En sentido vertical, los componentes están ordenados en paralelo, por lo que en cierto sentido, “ocupan el mismo espacio vertical”. Los componentes también se pueden ordenar secuencialmente en dirección vertical, y en paralelo en dirección horizontal, como veremos en la sección F.3. Para evitar traslapar los componentes, por lo general, los componentes con orientación vertical en paralelo tienen una orientación horizontal secuencial (y viceversa).

Grupos y alineación

Para crear interfaces de usuario más complejas, GroupLayout nos permite crear **grupos** que contengan elementos secuenciales o en paralelo. Dentro de un grupo, podemos tener componentes de GUI, otros grupos y huecos.

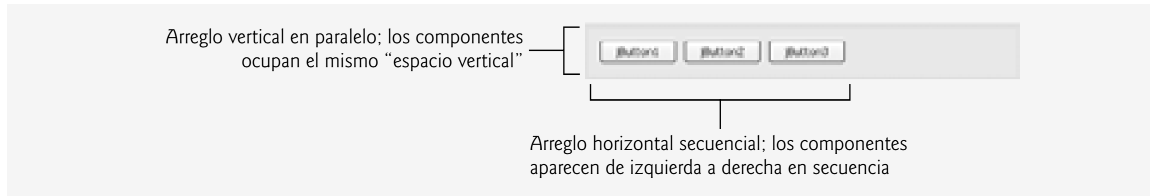


Figura F.1 | Objetos `JButton` ordenados en secuencia para su orientación horizontal, y en paralelo para su orientación vertical.

Colocar un grupo dentro de otro grupo es similar a crear una GUI usando contenedores anidados, como un objeto `JPanel` que contiene otros objetos `JPanel`, que a su vez contienen componentes de GUI.

Al crear un grupo, podemos especificar la **alineación** de sus elementos. La clase `GroupLayout` contiene cuatro constantes para este fin: `LEADING`, `TRAILING`, `CENTER` y `BASELINE`. La constante `BASELINE` se aplica sólo a orientaciones verticales. En la orientación horizontal, las constantes `LEADING`, `TRAILING` y `CENTER` representan la justificación a la izquierda, justificación a la derecha y centrado, respectivamente. En la orientación vertical, `LEADING`, `TRAILING` y `CENTER` alinean los componentes en su parte superior, inferior o centro vertical, respectivamente. Al alinear componentes con `BASELINE` estamos indicando que deben alinearse mediante el uso de la línea base de la fuente para el texto del componente. Para obtener más información acerca de las líneas base, vea la sección 12.4.

Espaciado

`GroupLayout` utiliza de manera predeterminada los lineamientos de diseño de GUIs de la plataforma subyacente para aplicar espacio entre un componente y otro. El método **addGap** de las clases de `GroupLayout` anidadas **`GroupLayout.Group`**, **`GroupLayout.SequentialGroup`** y **`GroupLayout.ParallelGroup`** nos permite controlar el espaciado entre componentes.

Ajustar el tamaño de los componentes

De manera predeterminada, `GroupLayout` utiliza los métodos `getMinimumSize`, `getMaximumSize` y `getPreferredSize` de cada componente para ayudar a determinar el tamaño del componente. Podemos redefinir la configuración predeterminada.

F.3 Creación de un objeto `SelectorColores`

Ahora vamos a presentar una aplicación llamada `SelectorColores` para demostrar el administrador de esquemas `GroupLayout`. Esta aplicación consiste en tres objetos `JSlider`, cada uno de los cuales representa los valores de `0` a `255` para especificar los valores rojo, verde y azul de un color. Los valores seleccionados para cada objeto `JSlider` se utilizarán para mostrar un rectángulo sólido del color especificado. Vamos a crear esta aplicación usando Netbeans 5.5. Para obtener una introducción más detallada acerca de cómo desarrollar aplicaciones de GUI en el IDE Netbeans, vea `www.netbeans.org/kb/trails/matisse.html`.

Cree un nuevo proyecto

Empiece por abrir un nuevo proyecto en Netbeans. Seleccione **File > New Project…**. En el cuadro de diálogo **New Project**, seleccione **General** de la lista **Categories** y **Java Application** de la lista **Projects**; después haga clic en **Next >**. Especifique `SelectorColores` como el nombre del proyecto y desactive la casilla de verificación **Create Main Class**. También puede especificar la ubicación de su proyecto en el campo **Project Location**. Haga clic en **Finish** para crear el proyecto.

Agregue una nueva subclase de JFrame al proyecto

En la ficha **Projects** del IDE, justo debajo del menú **File** y la barra de herramientas (figura F.2), expanda el nodo **Source Packages**. Haga clic con el botón derecho del ratón en el nodo **<default package>** que aparece y seleccione **New > JFrame Form**. En el cuadro de diálogo **New JPanel Form**, especifique `SelectorColores` como el nombre de la clase y haga clic en **Finish**. Esta subclase de `JFrame` mostrará los componentes de la GUI de la aplicación. La ventana de Netbeans ahora deberá ser similar a la figura F.3, mostrando la clase `SelectorColores` en vista de diseño (**Design**). Los botones **Source** y **Design** en la parte superior de la ventana `SelectorColores.java` nos permiten alternar entre editar el código fuente y diseñar la GUI.

La vista **Design** sólo muestra el área cliente de `SelectorColores` (es decir, el área que aparecerá dentro de los bordes de la ventana). Para crear una GUI en forma visual, puede arrastrar componentes de GUI desde la ventana **Palette** hacia el área cliente. Para configurar las propiedades de cada componente, hay que seleccionarlo y después modificar los valores de las propiedades que aparecen en la ventana **Properties** (figura F.3). Al seleccionar un componente, la ventana **Properties** muestra tres botones: **Properties**, **Events** y **Code** (vea la figura F.4); éstos le permiten configurar varios aspectos del componente.



Figura F.2 | Agregue un nuevo **formulario JFrame** al proyecto `SelectorColores`.

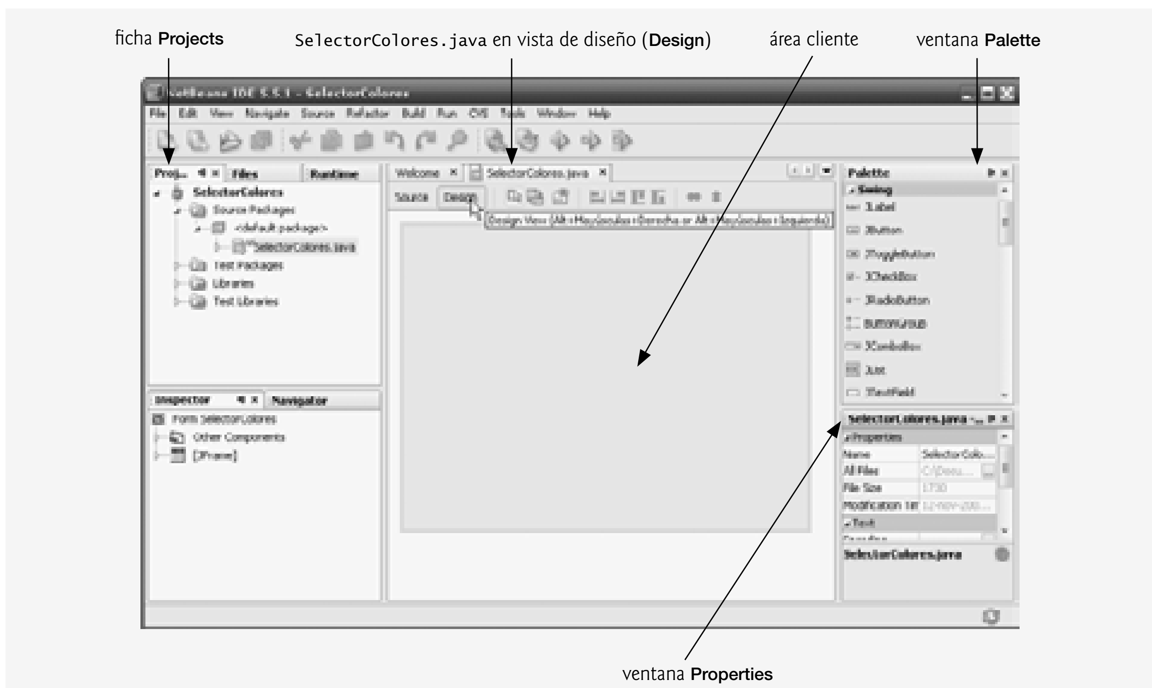


Figura F.3 | La clase `SelectorColores` se muestra en vista **Design** de Netbeans.

Cree la GUI

Arrastre tres componentes `JSlider` de la paleta (**Palette**) hacia el formulario `JFrame` (tal vez necesite desplazarse por la paleta). A medida que arrastramos componentes cerca de los bordes del área cliente, o cerca de otros componentes, Netbeans muestra **líneas guía** (figura F.4) que indican las distancias y alineaciones recomendadas entre el componente que estamos arrastrando, los bordes del área cliente y los demás componentes. A medida que siga los pasos para crear la GUI, use las líneas guía para ordenar los componentes en tres filas y tres columnas, como en la figura F.5. Use la ventana **Properties** para cambiar el nombre de los componentes `JSlider` a `rojoJSlider`, `verdeJSlider` y `azulJSlider`. Seleccione el primer componente `JSlider`, después haga clic en el botón **Code** de la ventana **Properties** y cambie la propiedad **Variable Name** a `rojoJSlider`. Repita este proceso para cambiar el nombre a los otros dos componentes `JSlider`. Después seleccione cada componente `JSlider` y cambie su propiedad `maximum` a 255, para que produzca valores en el rango de 0 a 255, y cambie su propiedad `value` a 0, de manera que el indicador del componente `JSlider` se encuentre inicialmente a la izquierda.

Arrastre tres componentes `JLabel` de la paleta al formulario `JFrame` para etiquetar cada componente `JSlider` con el color que representa. Use los nombres `rojoJLabel`, `verdeJLabel` y `azulJLabel` para los componentes `JLabel`, respectivamente. Cada componente `JLabel` debe colocarse a la izquierda del componente `JSlider` correspondiente (figura F.5). Cambie la propiedad **text** de cada componente `JLabel`, ya sea haciendo doble clic en el componente `JLabel` y escribiendo el nuevo texto, o seleccionando el componente `JLabel` y cambiando la propiedad **text** en la ventana **Properties**.

Agregue un componente `JTextField` a cada uno de los componentes `JSlider` para mostrar su valor. Use los nombres `rojoJTextField`, `verdeJTextField` y `azulJTextField` para estos componentes. Cambie la propiedad **text** de cada componente `JTextField` a `0`, usando las mismas técnicas que para los componentes `JLabel`. Cambie la propiedad **columns** de cada componente `JTextField` a `4`.



Figura F.4 | Posicione el primer componente `JTextField`.



Figura F.5 | Distribución de los componentes `JLabel`, `JSlider` y `JTextField`.

Haga doble clic en el borde del área cliente para que aparezca el cuadro de diálogo **Set Form Designer Size** y cambie el primer número (que representa la anchura) a 410; después haga clic en **OK**. Esto hace al área cliente lo suficientemente amplia como para poder alojar el componente `JPanel` que agregará a continuación. Por último, agregue un componente `JPanel` llamado `colorJPanel` a la derecha de este grupo de componentes. Use las líneas guía como se muestra en la figura F.6 para colocar el componente `JPanel`. Cambie el color de fondo de este componente para mostrar el color seleccionado. Por último, arrastre el borde inferior del área cliente hacia la parte superior del área **Design**, de manera que pueda ver la línea de ajuste que muestra la altura recomendada del área cliente (con base en sus componentes), como se muestra en la figura F.7.



Figura F.6 | Posicione el panel `JLabel`.

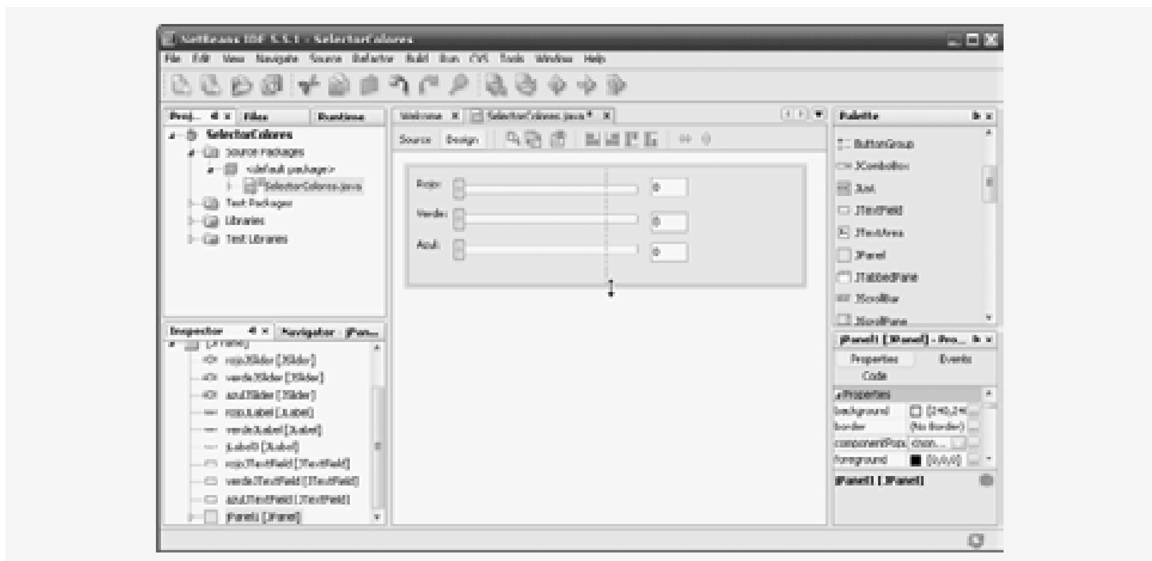


Figura F.7 | Ajuste de la altura del área cliente.

Edite el código fuente y agregue manejadores de eventos

El IDE generó de manera automática el código de la GUI, incluyendo métodos para inicializar componentes y alinearlos mediante el administrador de esquemas `GroupLayout`. Debemos agregar la funcionalidad deseada a los manejadores de eventos de los componentes. Para agregar un manejador de eventos para un componente, haga clic con el botón derecho sobre él y coloque el ratón sobre la opción **Events** en el menú contextual. A continuación, podrá seleccionar la categoría de evento que desee manejar, y el evento específico dentro de esa categoría. Por ejemplo, para agregar los manejadores de eventos para los componentes `JSlider` para este ejemplo, haga clic en cada componente `JSlider` y seleccione **Events > Change > stateChanged**. Al hacer esto, Netbeans agrega un objeto `ChangeListener` al componente `JSlider` y cambia de vista de diseño (**Design**) a vista de código fuente (**Source**), en donde podemos colocar código en el manejador de eventos. Use el botón **Design** para regresar a la vista de diseño y repita los pasos anteriores para agregar los manejadores de eventos para los otros dos componentes `JSlider`. Para completar los manejadores de eventos, agregue primero el método de la figura F.8. En cada manejador de eventos de `JSlider`, establezca el componente `JTextField` correspondiente con el nuevo valor del componente `JSlider`, y después llame al método `cambiarColor`. Por último, en el constructor después de la llamada a `initComponents`, agregue la línea:

```
colorJPanel.setBackground( java.awt.Color.BLACK );
```

La figura F.9 muestra la clase `SelectorColores` completa, exactamente como la genera Netbeans. Cada vez una mayor parte del desarrollo de software se lleva a cabo con herramientas que generan código complicado como éste, lo cual ahorra al lector el tiempo y esfuerzo de hacerlo por sí mismo.

```
// cambia el color de fondo del componente colorJPanel, con base en los valores
// actuales de los componentes JSlider
public void cambiarColor()
{
   colorJPanel.setBackground( new java.awt.Color(
      rojoJSlider.getValue(), verdeJSlider.getValue(),
      azulJSlider.getValue() ) );
} // fin del método cambiarColor
```

Figura F.8 | Método que cambia el color de fondo de `colorJPanel`, con base en los valores de los tres componentes `JSlider`.

```
/*
 * SelectorColores.java
 *
 * Created on 12 de noviembre de 2007, 3:51
 */

/**
 *
 * @author Administrador
 */
public class SelectorColores extends javax.swing.JFrame
{

   /** Creates new form SelectorColores */
   public SelectorColores()
   {
      initComponents();
      colorJPanel.setBackground( java.awt.Color.BLACK );
   }
```

Figura F.9 | Clase `SelectorColores` que utiliza a `GroupLayout` para su esquema de GUI. (Parte 1 de 5).

```java

   // cambia el color de fondo del componente colorJPanel, con base en los valores
   // actuales de los componentes JSlider
   public void cambiarColor()
   {
      colorJPanel.setBackground( new java.awt.Color(
         rojoJSlider.getValue(), verdeJSlider.getValue(),
         azulJSlider.getValue() ) );
   } // fin del método cambiarColor

   /** This method is called from within the constructor to
    * initialize the form.
    * WARNING: Do NOT modify this code. The content of this method is
    * always regenerated by the Form Editor.
    */
   // <editor-fold defaultstate="collapsed" desc=" Generated Code ">
   private void initComponents() {
      rojoJSlider = new javax.swing.JSlider();
      verdeJSlider = new javax.swing.JSlider();
      azulJSlider = new javax.swing.JSlider();
      rojoJLabel = new javax.swing.JLabel();
      verdeJLabel = new javax.swing.JLabel();
      jLabel3 = new javax.swing.JLabel();
      rojoJTextField = new javax.swing.JTextField();
      verdeJTextField = new javax.swing.JTextField();
      azulJTextField = new javax.swing.JTextField();
      colorJPanel = new javax.swing.JPanel();

      setDefaultCloseOperation(javax.swing.WindowConstants.EXIT_ON_CLOSE);
      rojoJSlider.setMaximum(255);
      rojoJSlider.setValue(0);
      rojoJSlider.setName("null");
      rojoJSlider.addChangeListener(new javax.swing.event.ChangeListener() {
         public void stateChanged(javax.swing.event.ChangeEvent evt) {
            rojoJSliderStateChanged(evt);
         }
      });

      verdeJSlider.setMaximum(255);
      verdeJSlider.setValue(0);
      verdeJSlider.setName("null");
      verdeJSlider.addChangeListener(new javax.swing.event.ChangeListener() {
          public void stateChanged(javax.swing.event.ChangeEvent evt) {
            verdeJSliderStateChanged(evt);
         }
      });

      azulJSlider.setMaximum(255);
      azulJSlider.setValue(0);
      azulJSlider.addChangeListener(new javax.swing.event.ChangeListener()
      {
         public void stateChanged(javax.swing.event.ChangeEvent evt) {
            azulJSliderStateChanged(evt);
         }
      });

      rojoJLabel.setText("Rojo:");

      verdeJLabel.setText("Verde:");
```

Figura F.9 | Clase `SelectorColores` que utiliza a `GroupLayout` para su esquema de GUI. (Parte 2 de 5).

```
        jLabel3.setText("Azul:");

        rojoJTextField.setColumns(4);
        rojoJTextField.setText("0");

        verdeJTextField.setColumns(4);
        verdeJTextField.setText("0");

        azulJTextField.setColumns(4);
        azulJTextField.setText("0");

        javax.swing.GroupLayout colorJPanelLayout = new javax.swing.GroupLayout(colorJPa-
        nel);
        colorJPanel.setLayout(colorJPanelLayout);
        colorJPanelLayout.setHorizontalGroup(
           colorJPanelLayout.createParallelGroup
           javax.swing.GroupLayout.Alignment.LEADING)
           .addGap(0, 100, Short.MAX_VALUE)
        );
        colorJPanelLayout.setVerticalGroup(
           colorJPanelLayout.createParallelGroup(
           javax.swing.GroupLayout.Alignment.LEADING)
           .addGap(0, 100, Short.MAX_VALUE)
        );

        javax.swing.GroupLayout layout = new
        javax.swing.GroupLayout(getContentPane());
        getContentPane().setLayout(layout);
        layout.setHorizontalGroup(
           layout.createParallelGroup(
javax.swing.GroupLayout.Alignment.LEADING)
           .addGroup(layout.createSequentialGroup()
              .addContainerGap()
              .addGroup(layout.createParallelGroup(
javax.swing.GroupLayout.Alignment.TRAILING, false)
                 .addComponent(rojoJLabel, javax.swing.GroupLayout.DEFAULT_SIZE, javax.
swing.GroupLayout.DEFAULT_SIZE, Short.MAX_VALUE)
                 .addComponent(verdeJLabel, javax.swing.GroupLayout.DEFAULT_SIZE, javax.
swing.GroupLayout.DEFAULT_SIZE, Short.MAX_VALUE)
                 .addComponent(jLabel3, javax.swing.GroupLayout.DEFAULT_SIZE, javax.
swing.GroupLayout.DEFAULT_SIZE, Short.MAX_VALUE))
                 .addPreferredGap(
javax.swing.LayoutStyle.ComponentPlacement.RELATED)
                 .addGroup(layout.createParallelGroup(
javax.swing.GroupLayout.Alignment.LEADING)
              .addGroup(layout.createSequentialGroup()
                 .addComponent(azulJSlider, javax.swing.GroupLayout.PREFERRED_SIZE,
javax.swing.GroupLayout.DEFAULT_SIZE, javax.swing.GroupLayout.PREFERRED_SIZE)
                 .addPreferredGap
javax.swing.LayoutStyle.ComponentPlacement.RELATED)
                 .addComponent(azulJTextField, javax.swing.GroupLayout.PREFERRED_SIZE,
javax.swing.GroupLayout.DEFAULT_SIZE, javax.swing.GroupLayout.PREFERRED_SIZE))
              .addGroup(layout.createSequentialGroup()
                 .addComponent(verdeJSlider, javax.swing.GroupLayout.PREFERRED_SIZE,
javax.swing.GroupLayout.DEFAULT_SIZE, javax.swing.GroupLayout.PREFERRED_SIZE)
                 .addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.RELATED)
                    .addComponent(verdeJTextField, javax.swing.GroupLayout.PREFERRED_
```

Figura F.9 | Clase `SelectorColores` que utiliza a `GroupLayout` para su esquema de GUI. (Parte 3 de 5).

```
SIZE, javax.swing.GroupLayout.DEFAULT_SIZE, javax.swing.GroupLayout.PREFERRED_SIZE))
                  .addGroup(layout.createSequentialGroup()
                     .addComponent(rojoJSlider, javax.swing.GroupLayout.PREFERRED_SIZE,
javax.swing.GroupLayout.DEFAULT_SIZE, javax.swing.GroupLayout.PREFERRED_SIZE)
                     .addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.RELATED)
                     .addComponent(rojoJTextField, javax.swing.GroupLayout.PREFERRED_
SIZE, javax.swing.GroupLayout.DEFAULT_SIZE, javax.swing.GroupLayout.PREFERRED_SIZE)))
                     .addPreferredGap(
javax.swing.LayoutStyle.ComponentPlacement.RELATED, 10, Short.MAX_VALUE)
                     .addComponent(colorJPanel, javax.swing.GroupLayout.PREFERRED_SIZE,
javax.swing.GroupLayout.DEFAULT_SIZE, javax.swing.GroupLayout.PREFERRED_SIZE)
                     .addContainerGap())
            );
            layout.setVerticalGroup(
               layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
               .addGroup(layout.createSequentialGroup()
                  .addContainerGap()
                  .addGroup(layout.createParallelGroup(
javax.swing.GroupLayout.Alignment.LEADING)
                     .addComponent(colorJPanel, javax.swing.GroupLayout.PREFERRED_SIZE,
javax.swing.GroupLayout.DEFAULT_SIZE, javax.swing.GroupLayout.PREFERRED_SIZE)
                     .addGroup(layout.createSequentialGroup()
                        .addGroup(layout.createParallelGroup(
javax.swing.GroupLayout.Alignment.LEADING, false)
                        .addGroup(layout.createSequentialGroup()
                          .addComponent(rojoJTextField, javax.swing.GroupLayout.
PREFERRED_SIZE, javax.swing.GroupLayout.DEFAULT_SIZE, javax.swing.GroupLayout.PREFERRED_SIZE)
                          .addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.
RELATED, javax.swing.GroupLayout.DEFAULT_SIZE, Short.MAX_VALUE)
                          .addComponent(verdeJTextField, javax.swing.GroupLayout.
PREFERRED_SIZE, javax.swing.GroupLayout.DEFAULT_SIZE, javax.swing.GroupLayout.PREFERRED_
SIZE))
                        .addGroup(layout.createSequentialGroup()
                          .addGroup(layout.createParallelGroup(
javax.swing.GroupLayout.Alignment.LEADING)
                          .addComponent(rojoJSlider, javax.swing.GroupLayout.PREFERRED_
SIZE, javax.swing.GroupLayout.DEFAULT_SIZE, javax.swing.GroupLayout.PREFERRED_SIZE)
                        .addComponent(rojoJLabel))
                          .addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.
RELATED)
                          .addGroup(layout.createParallelGroup(
javax.swing.GroupLayout.Alignment.LEADING)
                          .addComponent(verdeJSlider, javax.swing.GroupLayout.PREFERRED_
SIZE, javax.swing.GroupLayout.DEFAULT_SIZE, javax.swing.GroupLayout.PREFERRED_SIZE)
                        .addComponent(verdeJLabel))))
                          .addPreferredGap(
javax.swing.LayoutStyle.ComponentPlacement.RELATED)
                          .addGroup(layout.createParallelGroup(
javax.swing.GroupLayout.Alignment.LEADING)
                        .addComponent(azulJLabel)
                          .addGroup(layout.createParallelGroup(
javax.swing.GroupLayout.Alignment.TRAILING)
                          .addComponent(azulJTextField, javax.swing.GroupLayout.
PREFERRED_SIZE, javax.swing.GroupLayout.DEFAULT_SIZE, javax.swing.GroupLayout.PREFERRED_SIZE)
                          .addComponent(azulJSlider, javax.swing.GroupLayout.PREFERRED_
SIZE, javax.swing.GroupLayout.DEFAULT_SIZE, javax.swing.GroupLayout.PREFERRED_SIZE)))))
                          .addContainerGap(javax.swing.GroupLayout.DEFAULT_SIZE, Short.
MAX_VALUE))
         );
```

Figura F.9 | Clase `SelectorColores` que utiliza a `GroupLayout` para su esquema de GUI. (Parte 4 de 5).

```
      pack();
   }// </editor-fold>

   private void azulJSliderStateChanged(javax.swing.event.ChangeEvent evt) {
      azulJTextField.setText( "" + azulJSlider.getValue() );
      cambiarColor();
   }

   private void verdeJSliderStateChanged(javax.swing.event.ChangeEvent evt) {
      verdeJTextField.setText( "" + verdeJSlider.getValue() );
      cambiarColor();
   }

   private void rojoJSliderStateChanged(javax.swing.event.ChangeEvent evt) {
      rojoJTextField.setText( "" + rojoJSlider.getValue() );
      cambiarColor();
   }

   /**
      * @param args the command line arguments
      */
   public static void main(String args[]) {
      java.awt.EventQueue.invokeLater(new Runnable() {
         public void run() {
            new SelectorColores().setVisible(true);
         }
      });
   }

   // Variables declaration - do not modify
   private javax.swing.JSlider azulJSlider;
   private javax.swing.JTextField azulJTextField;
   private javax.swing.JPanel colorJPanel;
   private javax.swing.JLabel jLabel3;
   private javax.swing.JLabel rojoJLabel;
   private javax.swing.JSlider rojoJSlider;
   private javax.swing.JTextField rojoJTextField;
   private javax.swing.JLabel verdeJLabel;
   private javax.swing.JSlider verdeJSlider;
   private javax.swing.JTextField verdeJTextField;
   // End of variables declaration

}
```



Figura F.9 | Clase `SelectorColores` que utiliza a `GroupLayout` para su esquema de GUI. (Parte 5 de 5).

El método `initComponents` (líneas 36 a 159) fue generado completamente por Netbeans, con base en las interacciones del lector con el diseñador de GUIs. Este método contiene el código que crea y da formato a la GUI. En las líneas 38 a 93 se construyen e inicializan los componentes de la GUI. En las líneas 91 a 159 se especifica la distribución de esos componentes mediante el uso de `GroupLayout`. En las líneas 104 a 129 se especifica el grupo horizontal y en las líneas 130 a 157 se especifica el grupo vertical.

Agregamos en forma manual la instrucción que modifica el color de fondo del componente `colorJPanel` en la línea 18, y el método `cambiarColor` en las líneas 23 a 28. Cuando el usuario desplaza el indicador en uno de los componentes `JSlider`, el manejador de eventos de ese componente establece el texto en su correspondiente componente `JTextField` con el nuevo valor del componente `JSlider` (líneas 162, 167 y 172), después llama el método `cambiarColor` (líneas 163, 168 y 173) para actualizar el color de fondo del componente `colorJPanel`. El método `cambiarColor` obtiene el valor actual de cada componente `JSlider` (líneas 26 y 27), y utiliza estos valores como argumentos para el constructor de `Color` y crear un nuevo objeto `Color`.

F.4 Recursos Web sobre GroupLayout

`weblogs.java.net/blog/tpavek/archive/2006/02/getting_to_know_1.html`
Parte 1 del mensaje publicado en el blog sobre `GroupLayout` de Tomas Pavel; presenta las generalidades detrás de la teoría de `GroupLayout`.

`weblogs.java.net/blog/tpavek/archive/2006/03/getting_to_know.html`
Parte 2 del mensaje publicado en el blog sobre `GroupLayout` de Tomas Pavel; presenta una GUI completa, implementada con `GroupLayout`.

`wiki.java.net/bin/view/Javadesktop/GroupLayoutExample`
Proporciona una demostración de una Libreta de direcciones, de una GUI creada en forma manual con `GroupLayout`, con código fuente.

`java.sun.com/developer/technicalArticles/Interviews/violet_pavek_qa.html`
Artículo: “La siguiente ola de GUIs: el proyecto Matisse y el IDE Netbeans 5.0”, por Roman Strobl.

`www.netbeans.org/kb/50/quickstart-gui.html`
Tutorial: “Creación de GUIs en Netbeans 5.0”, por Talley Mulligan. Un recorrido a través de la creación de GUIs en Netbeans.

`testwww.netbeans.org/kb/41/flash-matisse.html`
Demostración en Flash del diseñador de la GUI Matisse de Netbeans, la cual utiliza a `GroupLayout` para ordenar componentes.

`www.developer.com/java/ent/article.php/3589961`
Tutorial sobre `GroupLayout` basado en Flash.

`weblogs.java.net/blog/claudio/archive/nb-layouts.html`
Tutorial: “Building Java GUIs with Matisse: A Gentle Introduction”, por Dick Wall.



Componentes de integración Java Desktop (JDIC)

G.1 Introducción

Los **Componentes de integración Java Desktop (JDIC)** son parte de un proyecto de código fuente abierto, orientado a permitir una mejor integración entre las aplicaciones de Java y las plataformas en las que se ejecutan. Algunas características de JDIC son:

- Interacción con la plataforma subyacente para iniciar aplicaciones nativas (como navegadores Web y clientes de correo electrónico).
- Mostrar una pantalla de inicio cuando una aplicación empieza a ejecutarse para indicar al usuario que se está cargando.
- Creación de iconos en la bandeja del sistema (también llamada área de estado de la barra de tareas, o área de notificación) para proporcionar acceso a las aplicaciones Java que se ejecutan en segundo plano.
- Registro de asociaciones de tipos de archivos, para que los archivos de tipos especificados se abran automáticamente en las correspondientes aplicaciones de Java.
- Creación de paquetes instaladores, y otras cosas más.

La página inicial de JDIC (`jdic.dev.java.net/`) incluye una introducción a JDIC, descargas, documentación, FAQs, demos, artículos, blogs, anuncios, proyectos Incubator, una página para el desarrollador, foros, listas de correo y mucho más. Java SE 6 ahora incluye algunas de las características antes mencionadas. Aquí hablaremos sobre varias de estas características.

G.2 Pantallas de inicio

Los usuarios de aplicaciones de Java perciben con frecuencia un problema en el rendimiento, ya que no aparece nada en la pantalla cuando se inicia una aplicación por primera vez. Una manera de mostrar a un usuario que su programa se está cargando es mediante una **pantalla de inicio**: una ventana sin bordes que aparece temporalmente mientras se inicia una aplicación. Java SE 6 proporciona la nueva opción de línea de comandos **`-splash`** para que el comando `java` pueda llevar a cabo esta tarea. Esta opción permite al programador especificar una imagen PNG, GIF o JPG que debe aparecer al momento en que una aplicación empieza a cargarse. Para demostrar esta nueva opción, creamos un programa (figura G.1) que permanece inactivo durante 5 segundos (para que el usuario pueda ver la pantalla de inicio) y después muestra un mensaje en la línea de comandos. El directorio para este ejemplo incluye una imagen en formato PNG para utilizarla como pantalla de inicio. Para mostrar la pantalla de inicio a la hora de cargar esta aplicación, use el siguiente comando:

```
java -splash:DeitelBug.png DemoSplash
```

```
// Fig. G.1: DemoInicio.java
// Demostración de la pantalla de inicio.
public class DemoInicio
{
   public static void main( String[] args )
   {
      try
      {
         Thread.sleep( 5000 );
      } // fin de try
      catch ( InterruptedException e )
      {
         e.printStackTrace();
      } // fin de catch

      System.out.println(
         "Esta fue la demostracion de la pantalla de inicio." );
   } // fin del método main
} // fin de la clase DemoInicio
```

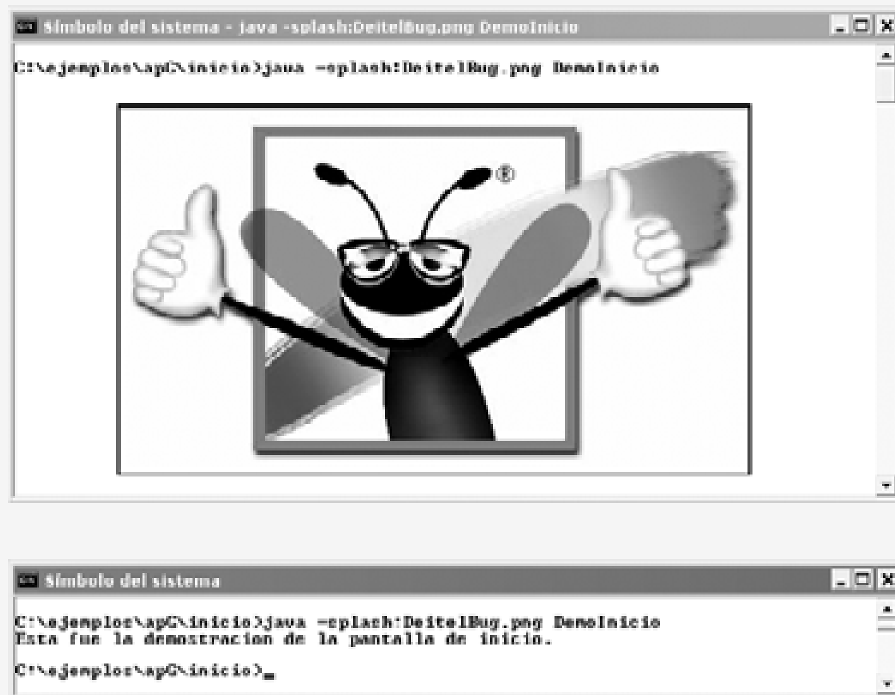


Figura G.1 | Pantalla de inicio que se muestra mediante la opción `-splash` del comando `java`.

Una vez que haya iniciado la visualización de la pantalla de inicio, podrá interactuar con ésta por medio de programación, mediante la clase `SplashScreen` del paquete `java.awt`. Para ello, puede agregar contenido dinámico a la pantalla de inicio. Para obtener más información acerca de cómo trabajar con las pantallas de inicio, vea los siguientes sitios:

```
java.sun.com/developer/technicalArticles/J2SE/Desktop/javase6/
   splashscreen/
java.sun.com/javase/6/docs/api/java/awt/SplashScreen.html
```

G.3 La clase Desktop

La nueva clase **Desktop** de Java SE 6 nos permite especificar un archivo o URI que deseemos abrir, mediante el uso de la aplicación apropiada de la plataforma subyacente. Por ejemplo, si el navegador Web predeterminado de su computadora es Firefox, puede usar el método `browse` de la clase `Desktop` para abrir un sitio Web en Firefox. Además, puede abrir una ventana de composición de correo electrónico en el cliente de correo electrónico predeterminado de su sistema, abrir un archivo en su aplicación asociada e imprimir un archivo mediante el uso del comando imprimir de la aplicación asociada. En la figura G.2 se demuestran las primeras tres de estas capacidades.

El manejador de eventos en las líneas 22 a 52 obtiene el número de índice de la tarea que el usuario selecciona en el componente `tareasJComboBox` (línea 25), y el objeto `String` que representa el archivo o URI a procesar (línea 26). En la línea 28 se utiliza el método `static` **isDesktopSupported** de `Desktop` para determinar si se soportan las características de la clase `Desktop` en la plataforma en la que se ejecute la aplicación. De ser así, en la línea 32 se utiliza el método `static` **getDesktop** de `Desktop` para obtener un objeto `Desktop`. Si el usuario seleccionó la opción para abrir el navegador Web predeterminado, en la línea 37 se crea un nuevo objeto `URI` mediante el uso del objeto `String` llamado `entrada` como el sitio a mostrar en el navegador, y después se pasa el objeto `URI` al método **browse** de `Desktop`, el cual invoca al navegador Web predeterminado del sistema y le pasa el `URI` para que lo muestre. Si el usuario selecciona la opción para abrir un archivo en su programa asociado, en la línea 40 se crea un nuevo objeto `File` usando el objeto `String` llamado **entrada** como el archivo a abrir, y después se pasa este objeto `File` al método **open** de `Desktop`, el cual pasa el archivo a la aplicación apropiada para que lo abra. Por último, si el usuario selecciona la opción para componer un correo electrónico, en la línea 43 se crea un nuevo objeto `URI` usando el objeto `String` llamado **entrada** como la dirección de correo a la cuál se enviará el correo electrónico, y después se pasa el objeto `URI` al método **mail** de `Desktop`, el cual invoca al cliente de correo electrónico predeterminado del sistema y pasa el `URI` a ese cliente de correo electrónico como el recipiente del mensaje. Para aprender más acerca de la clase `Desktop`, visite el sitio:

`java.sun.com/javase/6/docs/api/java/awt/Desktop.html`

```
1  // Fig. G.2: DemoDesktop.java
2  // Usa a Desktop para iniciar el navegador predeterminado, abrir un archivo en su
3  // aplicación asociada y componer un email en el cliente de email predeterminado.
4  import java.awt.Desktop;
5  import java.io.File;
6  import java.io.IOException;
7  import java.net.URI;
8
9  public class DemoDesktop extends javax.swing.JFrame
10 {
11    // constructor
12    public DemoDesktop()
13    {
14       initComponents();
15    } // fin del constructor de DemoDesktop
16
17    // Para ahorrar espacio, no mostramos aquí las líneas 20 a 84 del código de GUI
18    // generado por Netbeans de manera automática. El código completo para este ejemplo se
19    // encuentra en el archivo DemoDesktop.java en el directorio de este ejemplo.
20
21    // determina la tarea seleccionada y la lleva a cabo
22    private void hacerTareaJButtonActionPerformed(
23       java.awt.event.ActionEvent evt)
24    {
25       int indice = tareasJComboBox.getSelectedIndex();
26       String entrada = entradaJTextField.getText();
```

Figura G.2 | Use a `Desktop` para iniciar el navegador Web predeterminado, abrir un archivo en su aplicación asociada y componer un correo electrónico en el cliente de correo predeterminado. (Parte 1 de 3).

```
        if ( Desktop.isDesktopSupported() )
        {
           try
           {
              Desktop escritorio = Desktop.getDesktop();

              switch ( indice )
              {
                 case 0: // abre el navegador
                    escritorio.browse( new URI( entrada ) );
                    break;
                 case 1: // abre el archivo
                    escritorio.open( new File( entrada ) );
                    break;
                 case 2: // abre la ventana de composición de email
                    escritorio.mail( new URI( entrada ) );
                    break;
              } // fin de switch
           } // fin de try
           catch ( Exception e )
           {
              e.printStackTrace();
           } // fin de catch
        } // end if
     } // fin del método hacerTareaJButtonActionPerformed

     public static void main(String args[])
     {
        java.awt.EventQueue.invokeLater(
           new Runnable()
           {
              public void run()
              {
                 new DemoDesktop().setVisible(true);
              }
           }
        );
     } // fin del método main

     // Variables declaration - do not modify//GEN-BEGIN:variables
     private javax.swing.JLabel entradaJLabel;
     private javax.swing.JTextField entradaJTextField;
     private javax.swing.JButton hacerTareaJButton;
     private javax.swing.JLabel instruccionesJLabel;
     private javax.swing.JComboBox tareasJComboBox;
     // End of variables declaration//GEN-END:variables
  }
```

Figura G.2 | Use a `Desktop` para iniciar el navegador Web predeterminado, abrir un archivo en su aplicación asociada y componer un correo electrónico en el cliente de correo predeterminado. (Parte 2 de 3).

G.4 Iconos de la bandeja

Los **iconos de la bandeja** comúnmente aparecen en la bandeja de sistema de nuestro sistema, en el área de estado de la barra de tareas o en el área de notificación. Por lo general, proporcionan un acceso rápido a las aplicaciones que se ejecutan en segundo plano en el sistema del programador. Al posicionar el ratón sobre uno de estos iconos, aparece una barra de herramientas indicando qué aplicación representa el icono. Si hace clic en el icono, aparecerá un menú contextual con opciones para esa aplicación.



Figura G.2 | Use a `Desktop` para iniciar el navegador Web predeterminado, abrir un archivo en su aplicación asociada y componer un correo electrónico en el cliente de correo predeterminado. (Parte 3 de 3).

Las clases `SystemTray` y `TrayIcon` (ambas del paquete `java.awt`) nos permiten crear y administrar nuestros propios iconos de la bandeja, de una manera independiente a la plataforma. La clase **`SystemTray`** proporciona acceso a la bandeja del sistema de la plataforma subyacente; la clase consiste en tres métodos:

- El método `static` **`getDefaultSystemTray`** devuelve la bandeja del sistema.
- El método **`addTrayIcon`** agrega un nuevo objeto `TrayIcon` a la bandeja del sistema.
- El método **`removeTrayIcon`** elimina un icono de la bandeja del sistema.

La clase **`TrayIcon`** consiste en varios métodos que permiten a los usuarios especificar un icono, un cuadro de información sobre las herramientas y un menú contextual para el icono. Además, los iconos de la bandeja soportan objetos `ActionListener`, `MouseListener` y `MouseMotionListener`. Para aprender más acerca de las clases `SystemTray` y `TrayIcon`, visite:

```
java.sun.com/javase/6/docs/api/java/awt/SystemTray.html
java.sun.com/javase/6/docs/api/java/awt/TrayIcon.html
```

G.5 Proyectos JDIC Incubator

Los Proyectos JDIC Incubator son desarrollados, administrados y pertenecen a los miembros de la comunidad de Java. Estos proyectos se asocian (pero no se distribuyen con) JDIC. Los Proyectos Incubator pueden en algún momento dado formar parte del proyecto JDIC, una vez que se hayan desarrollado por completo y cumplan con ciertos criterios. Para obtener más información acerca de los Proyectos Incubator, y para aprender cómo puede establecer un Proyecto Incubator, visite el sitio:

`jdic.dev.java.net/#incubator`

Los Proyectos Incubator actuales son:

- FileUtil: la API de una herramienta de archivos, que extiende a la clase java.io.File.
- Floating Dock Top-level Window: una API de Java para desarrollar una ventana flotante acoplable de nivel superior.
- Icon Service: devuelve un objeto icono de Java, de una especificación de icono nativa.
- Misc API: Contiene APIs simples (un método, un tipo de clase).
- Music Player Control API: API de Java que controla los reproductores de música nativos.
- SaverBeans Screensaver SDK: kit de desarrollo de protectores de pantalla en Java.
- SystemInfo: comprueba la información del sistema.

G.6 Demos de JDIC

El sitio de JDIC incluye aplicaciones de demostración para FileExplorer, el paquete de navegador, el paquete TrayIcon, la clase Floating Dock y la API Wallpaper (`jdic.dev.java.net/#demos`). El código fuente para estos demos se incluye en la descarga de JDIC (`jdic.dev.java.net/servlets/ProjectDocumentList`). Para obtener más demos, dé un vistazo a algunos de los proyectos Incubator.



Mashups

Introducción

La creación de mashups de aplicaciones Web es uno de los temas insignia de Web 2.0. El término mashup se originó en el mundo de la música; un mashup es un remix de dos o más canciones para crear una nueva canción. Puede escuchar algunos mashups de música en `www.ccmixter.org/`. Un mashup de aplicación Web combina la funcionalidad complementaria que, por lo general, se utiliza a través de servicios Web (capítulo 28) y transmisiones RSS (`www.deitel.com/rss` y `www.rssbus.com`) de varios sitios Web. Podemos crear poderosas e innovadoras aplicaciones mashup Web 2.0 con mucha más rapidez que si tuviéramos que escribir las aplicaciones desde cero. Por ejemplo, `www.housingmaps.com` combina los listados de apartamentos Craigslist con Google Maps para mostrar en un mapa todos los apartamentos en renta en un vecindario.

Mashups populares

En la figura H.1 se muestran algunos mashups populares.

URL	APIs	Descripción
Mashups populares de Google Maps:		
`www.mappr.com/`	Google Maps, FlickR	Búsqueda de fotografías de sitios en EE.UU.
`www.housingmaps.com/`	Google Maps, Craigslist	Búsqueda de apartamentos y casas disponibles por vecindario. Incluye precios, fotografías, la dirección y la información de contacto del agente de bienes raíces.
`www.broadwayzone.com/`	Google Maps	Búsqueda de ubicaciones de teatros en la ciudad de Nueva York y los espectáculos de cada teatro. Vínculos a los detalles acerca del espectáculo, información sobre boletos e indicaciones para el metro.
`www.cribseek.com`	Google Maps	Mapas con propiedades en venta.
`www.shackprices.com/`	Google Maps	Búsqueda del valor aproximado de una casa, con base en las ventas recientes de casas en el área.
`www.mashmap.com/`	Google Maps	Haga clic en un cine en el mapa para buscar películas y mostrar horarios.

Figura H.1 | Mashups populares. (Parte 1 de 2).

URL	APIs	Descripción
`paul.kedrosky.com/publicloos/`	Google Maps	Búsqueda de sanitarios públicos en San Francisco. Incluye la dirección, una clasificación y comentarios sobre cada sanitario.
Otros mashups populares:		
`www.doubletrust.net`	Yahoo! Search, Google Search	Combina los resultados de búsquedas de Yahoo! y Google en una página.
`api.local.yahoo.com/eb/`	Yahoo! Maps	Búsqueda de la ubicación de ciertos eventos (por fecha) en un área geográfica.
`www.csthota.com/geotagr`	Microsoft Virtual Earth	Almacene y explore fotografías por ubicación geográfica.
`www.kokogiak.com/amazon4/default.asp`	Amazon Web Services	Agregue artículos de Amazon a su lista de regalos, coloque el vínculo a un libro en su blog en Blogger, agregue un vínculo a sus sitios favoritos `del.icio.us` o busque el libro en su biblioteca local.

Figura H.1 | Mashups populares. (Parte de 2 de 2).

Ahora que ha leído la mayor parte de este libro, tal vez esté familiarizado con las categorías de APIs, incluyendo gráficos, GUI, colecciones, multimedia, bases de datos y muchas más. Casi todas ellas proporcionan una *funcionalidad de cómputo* mejorada. Muchas APIs de servicios Web proporcionan *funcionalidad comercial*: eBay proporciona herramientas para subastas, Amazon proporciona ventas de libros (y ventas de otros tipos de productos, como CDs, DVDs, dispositivos electrónicos, y otros más), Google proporciona herramientas de búsqueda, PayPal proporciona servicios de pago, etcétera. Por lo general, estos servicios Web son gratuitos para su uso no comercial; algunos establecen cuotas (por lo general, razonables) para su uso comercial. Esto crea enormes posibilidades para las personas que crean aplicaciones y comercios basados en Internet.

APIs de uso común en mashups

Hemos enfatizado la importancia de la reutilización de software. Los mashups son otra forma más de reutilización de software que nos ahorra tiempo, dinero y esfuerzo; podemos crear rápidamente versiones prototipo de nuestras aplicaciones, integrar funcionalidad de negocios, de búsqueda y mucho más. En la figura H.2 se muestran algunas APIs de uso común en mashups.

Origen de API	URL	Funcionalidad
Google Maps	`www.google.com/apis/maps`	Mapas.
Yahoo! Maps	`developer.yahoo.net/maps/`	Mapas.
Microsoft Virtual Earth	`virtualearth.msn.com/`	Búsqueda local, mapas.
Amazon	`aws.amazon.com/`	Comercio electrónico.
TypePad ATOM	`www.sixapart.com/pronet/docs/typepad_atom_api`	Blogging.
Blogger ATOM feed	`code.blogspot.com`	Blogging.
Flickr	`developer.yahoo.net/flickr/index.html`	Compartir fotografías.
YouTube	`www.youtube.com/dev`	Compartir videos.
PayPal	`developer.paypal.com`	Pagos.

Figura H.2 | APIs de uso común para crear mashups. (Parte de 1 de 2).

Origen de API	URL	Funcionalidad
del.icio.us	`del.icio.us/help/api/`	Sitios favoritos de interés social.
Backpack	`backpackit.com/`	Programación de eventos.
Dropcash	`www.dropcash.com/`	Organizador de recaudación de fondos.
Upcoming.org	`upcoming.org/services/api/`	Listados de eventos de sindicatos.
Google AdWords	`www.google.com/apis/adwords/`	Administrar programas de publicidad de Google AdWords.
eBay	`developer.ebay.com/common/api`	Subastas.
SalesForce	`www.salesforce.developer/`	Administración de relaciones con los clientes (CRM).
Technorati	`developers.technorati.com/ wiki/TechoratiApi`	Búsqueda en Blogs.

Figura H.2 | APIs de uso común para crear mashups. (Parte de 2 de 2).

Centro de recursos Deitel sobre mashups

Nuestro Centro de recursos sobre mashups, que se encuentra en

`www.deitel.com/mashups/MashUpsResourceCenter.html`

se enfoca en la enorme cantidad de contenido de mashups gratuito, disponible en línea. Encontrará tutoriales, artículos, documentación, los libros más recientes, blogs, directorios, herramientas, foros, etcétera, que le ayudarán a desarrollar rápidamente aplicaciones de mashups.

- Dé un vistazo a los mashups más recientes y populares, incluyendo decenas de mashups basados en Google Maps, mostrando la ubicación de cines, bienes raíces para venta o renta, propiedades que se han vendido en su área, ¡e incluso las ubicaciones de los sanitarios públicos en San Francisco!
- Busque mashups en ProgrammableWeb por categoría.
- Dé un vistazo a las APIs de Flickr para agregar fotografías a sus aplicaciones, actualizar fotografías, reemplazarlas, ver peticiones de ejemplos y enviar en forma asíncrona.
- Lea el artículo "Building Mashups for Non-Programmers".
- Dé un vistazo a la herramienta Smashforce que permite a los usuarios de Salesforce.com usar aplicaciones como Google Maps con sus aplicaciones Multiforce y Sforce empresariales.
- Busque sitios sobre mashups tales como ProgrammableWeb, Givezilla, Podbop y Strmz.
- Dé un vistazo a la Herramienta de Mashups empresarial de IBM.
- Dé un vistazo a las APIs de búsqueda y mapas de Microsoft, Yahoo! Y Google que puede usar en sus aplicaciones de mashups.
- Use las APIs de Technorati para buscar todos los blogs que vinculen a un sitio específico, busque la mención de ciertas palabras en blogs, vea cuáles blogs están vinculados con un blog dado y busque blogs asociados con un sito Web específico.
- Use la API de Backpack para que le ayude a organizar tareas y eventos, planear su itinerario, colaborar con otros, monitorear a sus competidores en línea, y mucho más.

Centro de recursos Deitel sobre RSS

Las transmisiones RSS son también fuentes de información populares para los mashups. Para aprender más acerca de las transmisiones RSS, visite nuestro Centro de recursos de RSS en `www.deitel.com/RSS/`. Cada semana anunciamos el (los) Centro(s) de Recursos más reciente(s) en nuestro boletín de correo electrónico gratuito, *Deitel Buzz Online*:

`www.deitel.com/newsletter/subscribe.html`

Envíe sus sugerencias sobre Centros de recursos adicionales y mejoras a los Centros de recursos existentes a `deitel@deitel.com`. ¡Muchas gracias!

Cuestiones de rendimiento y confiabilidad de los mashups

Hay varios retos a vencer al crear aplicaciones de mashups. Sus aplicaciones se hacen susceptibles a los problemas de tráfico y confiabilidad en Internet; circunstancias que, por lo general, están fuera de su control. Las compañías podrían cambiar repentinamente las APIs que sus aplicaciones utilizan. Su aplicación depende de las herramientas de hardware y software de otras compañías. Además, las compañías podrían establecer estructuras de cuotas para servicios Web anteriormente gratuitos, o podrían incrementar las cuotas existentes.

Tutoriales sobre mashups

En ésta y en las siguientes secciones, listaremos una gran cantidad de recursos sobre mashups, de nuestro Centro de recursos sobre mashups. Una vez que haya dominado los servicios Web en el capítulo 28, encontrará que la creación de mashups es un proceso bastante simple. Para cada API que desee utilizar, sólo visite el sitio correspondiente, regístrese y obtenga su clave de acceso (si se requiere), dé un vistazo a las implementaciones de ejemplo y asegúrese de aceptar sus acuerdos de "condiciones del servicio".

`www.programmableweb.com/howto`

El tutorial "How to Make your Own Web Mashup", de Programmableweb.com, es un tutorial de 5 pasos para crear un mashup. Los temas incluyen seleccionar un asunto, buscar sus datos, analizar sus habilidades de codificación, registrarse para una API, y empezar a codificar. Incluye una lista de APIs disponibles.

`blogs.msdn.com/jhawk/archive/2006/03/26/561658.aspx`

El tutorial "Building a Mashup of National Parks Using the Atlas Virtual Earth Map Control" de Jonathan Hawkins le muestra cómo mostrar chinchetas en un mapa de Microsoft Virtual Earth. Incluye un breve recorrido de la aplicación y una guía paso a paso para crear esta aplicación (incluye código en C#).

`www-128.ibm.com/developerworks/edu/x-dw-x-ultimashup1.html?`
`ca=dgrlnxw07WebMashupsPart1&s_cmp=gr&s_tact=105agx59`

El tutorial "The Ultimate Mashup—Web Services and the Semantic Web" de seis partes, publicado por IBM, trata acerca del concepto de los mashups, crear una caché de XML, RDF, Web Ontology Language (OWL), control de usuario y mucho más. El tutorial es principalmente para empleados de IBM; otros deben registrarse.

`conferences.oreillynet.com/cs/et2005/view/e_sess/6241`

Descargue la presentación sobre Mashups de la Conferencia sobre tecnologías emergentes de O'Reilly.

`www.theurer.cc/blog/2005/11/03/how-to-build-a-maps-mash-up/`

Tutorial "How to Build a Maps Mashup", por Dan Theurer. Incluye el código de JavaScript y una aplicación mashup de ejemplo.

Directorios sobre mashups

`www.programmableweb.com/mashups`

ProgrammableWeb (`www.programmableweb.com/mashups`) lista los mashups y APIs más recientes, además de las noticias y desarrollos Web. Incluye un directorio de nuevos mashups, los mashups más populares y otras cosas más. Puede buscar mashups por etiquetas comunes, incluyendo mapas, fotografías, búsqueda, compras, deportes, viajes, mensajería, noticias, tránsito y bienes raíces. De un vistazo a la matriz de Web 2.0 Mashup, con vínculos a numerosos mashups. Para cada sitio, encontrará los mashups que se han creado con los otros sitios en la matriz.

`www.programmableweb.com/matrix`

ProgrammableWeb incluye una matriz Web 2.0 Mashup con vínculos a numerosos mashups. Para cada uno de los sitios listados, puede buscar las mashups que se hayan creado con otros sitios en la matriz.

`googlemapsmania.blogspot.com/`

Lista numerosos mashups que utilizan Google Maps. Algunos ejemplos incluyen mashups de Google Maps con hoteles en EE.UU., información de tránsito, noticias sobre el Reino Unido y mucho más.

`www.webmashup.com`

Un directorio abierto para mashups y APIs de Web 2.0.

Recursos sobre mashups

`code.google.com/`

Las APIs de Google incluyen a Google Maps, Google AJAX Search API, Google Toolbar API, AdWords API, Google Data APIs, Google Checkout API y WikiWalki (APIs de google utilizadas en Google Maps).

`www.flickr.com/services/api/`

Las APIs disponibles de Flickr incluyen la actualización de fotografías, reemplazo de fotografías, peticiones de ejemplo y envío asíncrono. Los kits de APIs incluyen Java, ActionScript, Cold Fusion, Common Lisp, cUrl, Delphi, .NET, Perl, PHO, PHP5, Python, REALbasic y Ruby.

`developers.technorati.com/wiki/TechnoratiApi`
Las APIs disponibles en Technorati incluyen CosmosQuery, SearchQuery, GetInfoQuery, OutboundQuery y BlogInfoQuery. Las APIs nuevas y experimentales incluyen TagQuery, AttentionQuery y KeyInfo.

`mashworks.net/wiki/Building_mashups_for_Non-Programmers`
Artículo "Building Mashups for Non-Programmers" de MashWorks. Proporciona fuentes a los no programadores para crear mashups, como vínculos a servicios de mapas y ejemplos de mashups creados por no programadores, mediante el uso de Google Maps y Flickr.

Herramientas y descargas sobre mashups

`mashup-tools.pbwiki.com/`
Mashup Tools Wiki es una fuente de herramientas y tips para el desarrollador, para crear mashups de tecnología.

`news.com.com/2100-1032_3.6046693.html`
Artículo: "Yahoo to Offer New Mashup Tools" por Anne Broache. Habla sobre el anuncio de Yahoo de que proporcionará APIs para crear mashups a través de su Red de desarrolladores. Además, Yahoo establecerá una Galería de aplicaciones para ver programas creados con las APIs.

`www.imediaconnection.com/content/10217.asp`
Artículo: "Marketing Mashup Tools" por Rob Rose. Habla acerca del uso de mashups para comercializar en sitios Web. Los temas incluyen sistemas de búsqueda en el sitio, administración de campañas de correo electrónico, sistemas de administración de contenido, sistemas para análisis de sitos Web y lo que hay que considerar al usar estas herramientas.

`datamashups.com/overview.html`
Herramienta gratuita: DataMashup.com es un servicio hospedado que ofrece una herramienta de código fuente abierto (AppliBuilder), la cual permite a los usuarios crear mashups. Hay una demo disponible.

`blogs.zdnet.com/Hinchcliffe/?p=63`
Blog: "Assembling Great Software: A Round-up of Eight Mashup Tools" por Dion Hinchcliffe. Habla acerca de lo que hacen los mashups, los sitios de origen de APIs tales como ProgrammableWeb, y su reseña de ocho herramientas de mashups, incluyendo Above All Studio (de Above All Software), Dapper (una herramienta para mashups en línea), DataMashups.com (excelente para mashups de aplicaciones pequeñas de negocios), JackBuilder (de JackBe): una herramienta de mashups basados en navegador, aRex (de Nexaweb), Process Engine (de Procession) para automatización de tareas, Ratchet-X Studio (de RatchetSoft) para la rápida integración de las aplicaciones, y RSSBus (de RSSBus) para crear mashups a partir de transmisiones RSS.

`www.ning.com`
Use esta herramienta gratuita para crear sus propias "aplicaciones sociales". Dé un vistazo a algunas de las aplicaciones que han creado las personas mediante el uso de Ning, incluyendo un mapa de las rutas de excursiones en el área de la bahía de San Francisco, reseñas de restaurantes con mapas, y mucho más. El cofundador de Ning fue Marc Andressen; uno de los fundadores de Netscape.

Artículos sobre mashups

`www.factiva.com/infopro/articles/Sept2006Feature.asp?node=menuElem1103`
Artículo: "Mashups—The API Buffer", de Factiva. Explica qué son los mashups y cómo se crean.

`www-128.ibm.com/developerworks/library/x-mashups.html?`
`ca=dgrlnxw16MashupChallenges`
Artículo: "Mashups: The New Breed of Web App: An Introduction to Mashups" por Duane Merrill. Habla sobre lo que son los mashups, los tipos de mashups (mapas, video, fotografía, búsqueda, compras y noticias), las tecnologías relacionadas (como la arquitectura, AJAX, protocolos Web, screen scraping, Web semántica, RDF, RSS y ATOM) y los desafíos técnicos y sociales.

`ajax.sys-con.com/read/203935.htm`

Artículo: "Mashup Data Formats: JSON versus XML/XMLHttpRequest" por Daniel B. Markham. Compara las tecnologías JSON (Notación de objetos JavaScript) y XML/XMLHttpRequest para utilizarlas en aplicaciones Web.

`www.techsoup.org/learningcenter/webbuilding/page5788.cfm`

Artículo: "Mashups: An Easy, Free Way to Create Custom Web Apps", por Brian Satterfield. Habla sobre los recursos para crear mashups. Lista varios sitios sobre mashups, incluyendo Givezilla (para fines sin lucro), Podbop (listados de conciertos y archivos MP3) y Strmz (video de flujo continuo, o "streaming video", blogs de video y podcasts de video).

`www.msnbc.msn.com/id/11569228/site/newsweek/`

Artículo: "Technology: Time For Your Mashup?" por N'gai Croal. Habla sobre la historia de los mashups, los mashups de música, de video y las aplicaciones Web.

`www.slate.com/id/2114791/`

Artículo sobre "newsmashing": un mashup de blogs con las historias de noticias a las cuales hacen referencia. Esto nos permite ver un artículo completo y leer comentarios relacionados de la blogósfera.

`images.businessweek.com/ss/05/07/mashups/index_01.htm`

Artículo de Business Week Online titulado "Sampling the Web's Best Mashups", en el cual se listan mashups populares.

`www.usatoday.com/tech/columnist/kevinmaney/2005-08-16-maney-google-mashups_x.htm`

Artículo que habla sobre la proliferación de los mashups de Google Maps.

`www.clickz.com/experts/brand/brand/article.php/3528921`

Artículo titulado "The Branding and Mapping Mashup". Habla sobre cómo se utilizan los mashups para llevar marcas a los usuarios con base en su ubicación. Por ejemplo, los usuarios pueden buscar la gasolina más económica en su área.

`www.usernomics.com/news/2005/10/mash-up-apps-and-competitive-advantage.html`

Artículo: "Mashup Apps and Competitive Advantage: Benefits of mashups including user experience".

Mashups en la blogósfera

`web2.wsj2.com/the_web_20_mashup_ecosystem_ramps_up.htm`

En el Blog Web 2.0 de Dion Hinchcliffe (presidente y CTO de Hinchcliffe & Company) se habla sobre los mashups. Incluye un excelente gráfico del Ecosistema de Mashups.

`www.techcrunch.com/2005/10/04/ning-launches/`

Blog que da seguimiento a las compañías y noticias sobre Web 2.0. Estos mensajes publicados hablan acerca de Ning, una herramienta gratuita que podemos usar para crear aplicaciones sociales.

`www.engadget.com/entry/1234000917034960/`

Aprenda a crear sus propios mashups de Google Maps.

`blogs.zdnet.com/web2explorer/?p=16&part=rss&tag=feed&subj=zdblog`

Mensaje publicado en el blog de ZDNet, titulado "Fun with mashups". Incluye vínculos a varios mashups.

FAQs y grupos de noticias sobre mashups

`groups.google.com/group/Google-Maps-API?lnk=gschg&hl=en`

Grupo de noticias de la API de Google Maps en Google Groups. Converse con otros desarrolladores sobre el uso de la API de Google Maps, obtenga respuestas a sus preguntas y comparta sus aplicaciones con otros.

`programmableweb.com/faq`

La FAQ sobre los mashups en ProgrammableWeb proporciona una introducción a los mashups y las APIs, y habla acerca de cómo crear sus propias mashups, además de otros temas.

Índice

T

U

V

W

X

Acuerdo de licencia y garantía limitada

El software se proporciona "COMO ESTÁ", sin garantía de ninguna clase. Ni los autores, ni los desarrolladores de software, ni Pearson Educación hacen ninguna representación o garantía, ya sea expresa o implícita, con respecto a los programas de software, su calidad, precisión o adecuación para un propósito específico. Por lo tanto, ni los autores, desarrolladores de software, ni Pearson Educación tendrán responsabilidad alguna para con usted o cualquier otra persona o entidad con respecto a cualquier pérdida o daños, supuestos o reales, ocasionados directa o indirectamente por los programas contenidos en el CD. Esto incluye, pero no se limita a, la interrupción del servicio, pérdida de datos, pérdida de tiempo en el salón de clases, pérdida de utilidades por consultoría o de otro tipo, o daños consecuentes derivados del uso de estos programas. Si el CD en sí está defectuoso, usted podrá devolverlo para obtener un reemplazo. El uso de este software está sujeto a las cláusulas y condiciones de la Licencia del Código Binario que se incluye en este libro. Lea las licencias cuidadosamente.

Al abrir este paquete, usted está de acuerdo en regirse por las cláusulas y condiciones de estas licencias. Si no está de acuerdo, no abra el paquete.

Uso del CD-ROM

La interfaz con el contenido del CD de Microsoft® Windows® está diseñada para iniciar automáticamente, a través del archivo AUTORUN.EXE. Si no aparece una pantalla de inicio automáticamente cuando usted inserte el CD en su computadora, haga clic en el archivo welcome.htm para iniciarlo de manera manual, o consulte el archivo readme.txt.

Requerimientos de software y hardware del sistema

- Procesador Pentium III a 500 MHz (mínimo) o superior; Sun® Java™ Studio Creator 2 Update 1 requiere un procesador Intel Pentium 4 de 1 GHz (o su equivalente).
- Microsoft Windows Server 2003, Windows XP (con Service Pack 2), Windows 2000 Professional (con Service Pack 4).
- 512 MB de RAM como mínimo; Sun Java Studio Creator 2 Update 1 requiere 1 GB de RAM
- 1 GB, mínimo, de espacio en el disco duro.
- Unidad de CD-ROM.
- Conexión a Internet.
- Explorador Web, Adobe® Acrobat® Reader® y una herramienta para descomprimir archivos zip.

NOTAS

NOTAS

NOTAS

NOTAS

Material en el CD

El material correspondiente a los capítulos 26 a 30 así como el de los apéndices, podrá encontrarlo en el CD que acompaña a este libro, en formato PDF, totalmente en español.